Estudo de Técnicas de Teste de Regressão Baseado em Mutação Seletiva

Luciana Andréia Fondazzi Martimiano

Orientação: Prof. Dr. José Carlos Maldonado

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação - USP, para obtenção do título de Mestre em Ciências - Área de Ciências de Computação e Matemática Computacional.

USP - São Carlos
1999

Capítulo 1

Introdução

1.1 Contexto

Com a crescente demanda de software e a conseqüente evolução da Engenharia de Software, atividades agregadas sob o nome de Garantia de Qualidade de Software têm sido introduzidas ao longo de todo o processo de desenvolvimento, entre elas as atividades de VV&T – Verificação, Validação e Teste, com o intuito de auxiliar na melhoria da qualidade e da produtividade.

As atividades de teste envolvem basicamente quatro etapas: planejamento de testes, projeto de casos de teste, execução e avaliação dos resultados (Myers, 1979; Biezer, 1990; Maldonado, 1991; Pressman, 1992). Essas atividades são realizadas em três diferentes níveis: nível de unidade, nível de integração e nível de sistema. Geralmente, os critérios de teste de software são estabelecidos a partir de três técnicas: **funcional**, **estrutural** e **baseada em erros**. Cada uma dessas técnicas estabelece os requisitos de teste a partir de diferentes aspectos do software. A técnica funcional estabelece os requisitos de teste a partir da especificação do programa. A técnica estrutural estabelece os requisitos a partir da implementação do programa, enquanto que a técnica baseada em erros estabelece os requisitos de teste a partir de erros típicos cometidos no processo de desenvolvimento de software. Independentemente da técnica utilizada, um ponto importante que se considera em diversos estudos empíricos é a redução dos custos aplicados às atividades de teste. Além da redução dos custos, a eficácia também é um dos aspectos fundamentais bastante investigado (Mathur e Wong, 1993; Wong, 1993; Mathur e Wong, 1994; Wong *et al.*, 1994a; Wong *et al.*, 1994b; Offut *et al.*, 1996a; Offut *et al.*, 1996b).

Observa-se que o conjunto de informações oriundas das atividades de teste pode ser utilizado no contexto das atividades de depuração, estimativa de confiabilidade e de manutenção (evolução) de software (Ostrand e Weyuker, 1988; Hartmann e Robson, 1990; Pressman, 1992; Varadan, 1995).

Independentemente da qualidade da concepção, desenvolvimento e teste do sistema antes de ter sido liberado, o produto de software irá certamente ser modificado por diversas razões, por exemplo, para atender às mudanças nas especificações e expectativas dos usuários (Lehman, 1996, Wong *et al.*, 1997a, Wong *et al.*, 1997b). Assim, a fase de manutenção é a etapa do ciclo de vida do software na qual são efetuadas alterações no produto após sua liberação para o usuário. Qualquer que seja o tipo de manutenção – corretiva, evolutiva, adaptativa ou preventiva - algumas tarefas comuns devem ser efetuadas: o entendimento, a modificação e a revalidação do software. Durante a etapa de revalidação deve-se demonstrar que não somente a nova lógica está correta, mas também que as partes do software que não foram modificadas permanecem intactas e o software como um todo ainda funciona corretamente (Pressman, 1992).

A atividade de teste da evolução do sistema é geralmente denominada de **teste de regressão** e visa a fornecer evidências de que as mudanças ocorridas não afetam adversamente as características previamente existentes, assim como a evidenciar que as eventuais novas funcionalidades estão de acordo com as expectativas do usuário, ou seja, visa a dar evidências de que o software funciona corretamente após a modificação. A principal diferença entre o teste de regressão e os testes realizados durante o desenvolvimento do software é que durante o teste de regressão pode-se ter, de antemão, um conjunto de casos de teste disponível para reutilização. Esse conjunto contribui para verificar se novos erros não foram introduzidos com as modificações realizadas.

Recentemente, diversos estudos têm sido realizados abordando o teste de regressão de software. Leung e White (1991) identificam duas abordagens para o teste de regressão: retestar-tudo (*retest-all*) e seletiva. Na abordagem *retest-all* todos os casos de teste utilizados durante a fase de desenvolvimento são empregados. Na abordagem seletiva um subconjunto de casos de teste é selecionado a partir do conjunto original identificando partes do programa modificado que devem ser testadas. Ainda, segundo Leung e White, as técnicas de teste de regressão devem ser aplicadas a um baixo custo sem que sua eficácia seja comprometida. Considerando essas duas abordagens, diversas técnicas de teste de regressão têm sido propostas na literatura (Ostrand e Weyuker, 1988; Hartmann e Robson, 1990; Bates e Horwitz, 1993; Chen *et al.*, 1994; Binkley, 1995; Forgács e Takács, 1997; Rosenblum e Weyuker, 1997; Rothermel e Harrold, 1997; Wong

et al., 1997a; Wong *et al.*, 1997b; Granja e Jino, 1999). Essas técnicas e diversas outras existentes na literatura contribuem para a redução dos custos e esforços que são aplicados durante a realização das atividades de teste de regressão.

Muitas das técnicas de teste de regressão utilizam mecanismos, técnicas e critérios oriundos da atividade de teste de software realizada durante o processo de desenvolvimento. São exemplos dessas técnicas: **Técnica baseada em Fluxo de Dados** (Ostrand e Weyuker, 1988) e **Técnica baseada em Mutação Seletiva** (Wong *et al.*, 1997b). Outros exemplos de técnicas de teste de regressão são: **Técnica baseada em Execução Simbólica** (Yau e Kishimoto, 1987), **Técnica baseada em Domínio** (Mayrhauser *et al.*, 1994), **Técnica baseada em Modificação** (Wong *et al.*, 1997a), **Técnica baseada em Cobertura** (Rothermel e Harrold, 1997), **Técnica baseada na Cobertura dos Critérios Potenciais-Usos** (Granja, 1997).

A importância de se reduzir o tamanho do conjunto de casos de teste de regressão e de aumentar a eficácia da detecção de defeitos em programas modificados é ressaltada por Harrold e Wong *et al.* (1993; 1997a). Em estudos empíricos realizados por Wong *et al.* (1997a) uma técnica híbrida que combina minimização e seleção baseada em prioridade é proposta. Basicamente, essa técnica identifica um subconjunto representativo de todos os casos de teste que resultam em diferentes comportamentos no programa modificado.

Além da técnica híbrida, Wong *et al.* (1997b) propõem uma técnica para minimização dos custos da aplicação dos testes de regressão. Basicamente, a técnica utiliza mutação seletiva por meio de um conjunto de operadores de mutação[1], visando a encontrar um conjunto minimizado de casos de teste a ser aplicado durante os testes de regressão. Nessa técnica, a característica mais importante é examinar como um critério de teste pode ser utilizado no teste de regressão a fim de ajudar os testadores a determinar quais casos de teste devem ser selecionados ou ter uma maior prioridade para o processo de revalidação das novas funcionalidades.

Diante dessa diversidade de técnicas, Rothermel e Harrold (1996) definiram um *framework* que estabelece características para a comparação e avaliação de técnicas seletivas. Além desse *framework*, Leung e White (1991) definiram um modelo de custo que compara os custos gastos em aplicar as abordagens de teste de regressão *retest*-all e seletiva. Tanto o *framework* quanto o modelo de custo auxiliam testadores a decidir qual abordagem e qual técnica de teste de regressão utilizar. Assim, este trabalho está inserido no contexto de estudos

[1] Introduzem pequenos erros sintáticos no programa.

empíricos para contribuir na análise, comparação e escolha de técnicas de teste de regressão utilizando o *framework* definido por Rothermel e Harrold.

1.2 Motivação

O Grupo de Engenharia de Software do Instituto de Ciências Matemáticas e de Computação - ICMC/USP, em colaboração com o Grupo de Engenharia de Software da Faculdade de Engenharia Elétrica da UNICAMP, tem desenvolvido pesquisas na área de teste, com ênfase em estudos teóricos e empíricos e no desenvolvimento de ferramentas de teste. As ferramentas desenvolvidas - *POKETOOL* (Chaim, 1991), *PROTEUM* (Delamaro, 1993) e *PROTEUM/IM* (Delamaro, 1997) - possibilitam a realização de trabalhos comparativos entre critérios de teste funcionais, estruturais e baseados em erros. Na linha de teste de regressão, procurando explorar o conhecimento e a experiência do grupo na atividade de teste, iniciou-se uma cooperação com o Dr. Eric Wong tendo sido definida uma estratégia de revalidação com base no critério Análise de Mutantes (Wong *et al.*, 1997b) que motiva o presente trabalho.

Estudos demonstram que mais de 2/3 dos custos durante o ciclo de vida de um software são gastos com a atividade de manutenção e, uma grande porcentagem disto é gasta para a revalidação do software (teste de regressão) (Myers, 1979; Pressman, 1992). Segundo Ostrand (1988), a probabilidade de erros serem introduzidos durante a fase de manutenção está entre 50 e 80%, tornando, assim, a atividade de teste de regressão extremamente importante.

A exemplo do que ocorre na atividade de teste, durante o desenvolvimento de software, várias restrições são impostas à atividade de teste de regressão: custo, tempo, pressões de mercado, e outras. Os custos associados ao teste de regressão são usualmente altos e uma questão pertinente que se coloca é "**Como selecionar um subconjunto de casos de teste que diferencie o programa original do programa modificado, ou seja, que revele os possíveis defeitos existentes no programa modificado**?", já que à medida que o software evolui, o conjunto de casos de teste aumenta e, conseqüentemente o custo do teste de regressão. Repetir todos os casos de teste é, em geral, impraticável e eliminar arbitrariamente casos de teste é correr riscos. Nessa perspectiva, vários pesquisadores têm investigado técnicas alternativas para selecionar um conjunto de casos de teste de regressão a partir dos casos de teste disponíveis para o programa original (Ostrand e Weyuker, 1988; Bates e Horwitz, 1993; Chen *et al.*, 1994; Binkley, 1995; Rothermel e Harrold, 1997; Wong *et al.*, 1997a; Wong *et al*, 1997b; Granja e Jino, 1999), e são

classificadas segundo Wong *et al.* (1997a), em duas abordagens: *baseadas no comportamento* e *baseadas na cobertura*.

As técnicas baseadas no comportamento selecionam casos de teste que causam um comportamento diferente entre o programa original e o programa modificado. Por outro lado, as técnicas baseadas na cobertura selecionam casos de teste que cobrem componentes modificados e componentes afetados pelas modificações. A diferença de abordagem origina diferentes resultados na seleção dos casos de teste.

1.3 Objetivos

Considerando os conceitos apresentados acima, como a importância das atividades de teste de software durante o processo de desenvolvimento de software, a importância das atividades do teste de regressão durante a evolução/manutenção do software e a necessidade de minimização dos custos dessas atividades, este trabalho tem por objetivo realizar estudos empíricos e comparativos utilizando as técnicas de teste de regressão propostas por Wong *et al.* (1997a, 1997b): **Técnica baseada em Modificação** e **Técnica baseada em Mutação Seletiva**. A Figura 1.1 ilustra as atividades realizadas durante este trabalho e as atividades realizadas por Wong *et al*. Essas atividades estão caracterizadas na figura por setas vermelhas contínuas e setas azuis tracejadas, respectivamente.

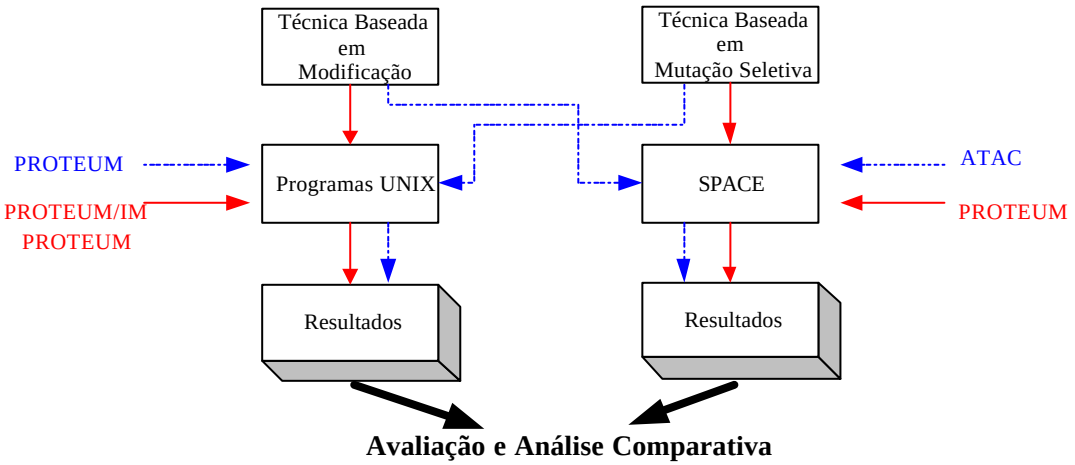

Figura 1.1 - Ilustração das atividades realizadas.

Procurar-se-á complementar e comparar os experimentos realizados por Wong *et al.* para avaliar ambas as técnicas de teste de regressão, gerando conhecimento e experiência na

perspectiva do estabelecimento de estratégias de revalidação eficazes e de baixo custo. Além dos estudos empíricos realizados, uma análise entre as técnicas de teste de regressão é realizada utilizando o *framework* desenvolvido por Rothermel e Harrold (1996).

1.4 Organização do Trabalho

Este capítulo apresentou o contexto no qual o trabalho está inserido, a motivação para realizá-lo e os objetivos a serem atingidos. O Capítulo 2 apresenta uma revisão bibliográfica pertinente aos principais conceitos relacionados a este trabalho, enfatizando os aspectos de evolução/manutenção de software e do teste de regressão. No Capítulo 3 são descritos os experimentos realizados com os programas utilitários do UNIX para avaliar a **Técnica baseada em Modificação** e os experimentos realizados com o programa SPACE para avaliar a **Técnica baseada em Mutação Seletiva**. O Capítulo 4 apresenta as contribuições deste trabalho e propostas de trabalhos futuros. O Apêndice A apresenta o conjunto de operadores de mutação utilizado e o Apêndice B apresenta as versões modificadas para os programas utilitários do UNIX e o processo de ativação dos erros do programa SPACE.

Capítulo 2

Revisão Bibliográfica

2.1 Considerações Iniciais

Neste capítulo são apresentados os principais conceitos pertinentes a este trabalho. Primeiramente, são apresentadas a importância da atividade de teste, suas principais técnicas e critérios. Em seguida, são apresentados os principais conceitos relacionados à manutenção de software e à atividade de teste de regressão. As técnicas de teste de regressão pertinentes a este trabalho são descritas, além do *framework* de avaliação de Rothermel e Harrold (1996) dessas técnicas e o modelo de custo de Leung e White (1991). Ferramentas de teste e estudos empíricos relacionados a este trabalho também são apresentados sucintamente.

2.2 A Atividade de Teste de Software

Segundo Myers (1979), o principal objetivo do teste de software é revelar a presença de erros no produto. Idealmente, o programa deveria ser exercitado com todos os valores do domínio de entrada possíveis. Sabe-se, entretanto, que o teste exaustivo é, em geral, impraticável devido às restrições de tempo e custo para realizá-lo.

Myers afirma que:

- Teste é um procedimento de executar um programa com a intenção de encontrar erros.
- Um bom caso de teste é aquele com alta probabilidade de encontrar erros.
- Um caso de teste bem sucedido é aquele que revela um erro ainda não descoberto.

Assim, o objetivo é determinar quais casos de teste devem ser utilizados de modo que a maioria dos erros existentes possa ser encontrada e que o número de casos de teste utilizado não seja tão grande a ponto de ser impraticável.

Critérios de teste têm sido elaborados com o objetivo de fornecer uma maneira sistemática e rigorosa para selecionar um subconjunto do domínio de entrada e ainda assim ser eficiente para revelar os erros existentes, respeitando as restrições de tempo e custo associados a um projeto de software. Esses critérios são classificados, basicamente, em três técnicas de teste: **Técnica Funcional**, **Técnica Estrutural** e a **Técnica Baseada em Erros**.

2.2.1 Técnica Funcional

A técnica funcional, também conhecida como **teste de caixa preta,** trata o software como uma caixa cujo conteúdo é desconhecido, sendo possível visualizarem-se somente os dados de entrada e as respostas produzidas como saída. Nessa técnica são verificadas as funções do sistema sem se preocupar com detalhes de implementação. Para isto, Coward (1988) distingue nessa técnica dois passos principais: primeiro, identificar as funções esperadas do software e segundo, criar casos de teste que chequem a corretitude dessas funções.

Um componente importante do teste funcional é o **oráculo**. O **oráculo** determina se a saída obtida, após a execução de uma função com os casos de teste, é correta ou não. A partir de ferramentas que simulem a especificação do sistema é possível obterem-se as informações necessárias para o oráculo; entretanto, na maioria das vezes, o oráculo é a figura do próprio testador (Howden, 1985; Biezer, 1990).

Os critérios de teste funcional mais conhecidos são: **Particionamento em Classes de Equivalência**, **Análise do Valor Limite**, **Grafos de Causa e Efeito** e ***Error Guessing***.

Particionamento em classes de equivalência divide o domínio de entrada de um programa em classes, a partir das quais os casos de teste podem ser derivados. O objetivo é minimizar o número de casos de teste, obtendo-se somente casos de teste essenciais, ou seja, que possuam alta probabilidade de revelar a presença dos erros existentes no programa. O uso de particionamento permite examinar os requisitos com mais detalhes e restringir o número de casos de teste existentes.

Análise do valor limite procura testar os limites das condições de entrada. Segundo Pressman (1992), erros costumam ocorrer com mais freqüência nos limites dos domínios de entrada do que no centro desses domínios, tornando o critério Análise de valor limite relevante para o teste funcional de um sistema.

Grafo de causa e efeito é utilizado para testar o efeito combinado de dados de entrada. Causas e efeitos são identificados e combinados em um grafo a partir do qual uma tabela de decisão é criada. Os dados de teste e as saídas são derivados da tabela de decisão obtida.

Error guessing consiste basicamente em listar possíveis erros e construir casos de teste associados a essas condições de erros.

Em geral, o teste funcional é uma técnica de validação de programas na qual os casos de teste são gerados a partir da especificação dos requisitos tornando-se, assim, uma técnica sujeita às inconsistências que podem ocorrer na especificação (DeMillo, 1987).

2.2.2 Técnica Estrutural

Essa técnica, também conhecida como **teste de caixa branca**, estabelece os requisitos de teste baseando-se nos aspectos de implementação, ou seja, no conhecimento da estrutura interna do programa. A maioria dos critérios dessa técnica utiliza uma representação de programa conhecida como **grafo de fluxo de controle** (GFC) ou **grafo de programa**. Um GFC é um grafo orientado no qual cada vértice representa um bloco indivisível de comandos e cada aresta representa um desvio de um bloco para outro. Um bloco desse tipo tem as seguintes características: não existem desvios para o meio do bloco e uma vez que o primeiro comando do bloco seja executado, todos os demais comandos do bloco são executados seqüencialmente. Utilizando-se do grafo de programa, os componentes que devem ser executados são escolhidos, caracterizando assim o teste estrutural.

Os critérios estruturais baseiam-se em tipos de estruturas diferentes para determinar quais partes do programa são requeridas na execução. Como critérios de teste estrutural tem-se: **Critérios Baseados em Fluxo de Controle, Baseados em Fluxo de Dados** e **Baseados na Complexidade**.

Os critérios baseados em fluxos de controle utilizam-se de características de controle da execução do programa, como comandos ou desvios, para determinar quais estruturas são necessárias. Os mais conhecidos são os **critérios Todos-Arcos, Todos-Nós** e **Todos-Caminhos**. Os critérios Todos-Nós e Todos-Arcos exigem que cada comando e cada aresta (cada desvio do programa) do GFC sejam exercitados pelo menos uma vez. Já o critério Todos-Caminhos que, geralmente, é impraticável, requer que todos os caminhos possíveis do programa sejam executados (Pressman, 1992). Um caminho (completo) é executável se existe um conjunto de valores que possa ser atribuído às variáveis de entrada do programa e que causa a execução desse caminho, caso contrário, esse caminho é dito não executável (Frankl, 1987).

Os critérios baseados em fluxo de dados utilizam-se de informações do fluxo de dados do programa para estabelecer os requisitos de teste. Esses critérios requerem que sejam testadas as interações que envolvam definições de variáveis e referências a essas definições. Exemplos dessa classe de critérios são os **Critérios de Rapps & Weyuker** (Rapps e Weyuker, 1982; Rapps e Weyuker, 1985) e os **Critérios Potenciais-Usos** (Maldonado, 1991).

Os critérios baseados na complexidade utilizam-se de informações sobre a complexidade do programa para determinar os requisitos de teste. Um critério bastante conhecido dessa classe é o **critério de McCabe** que utiliza a **complexidade ciclomática**[2] para estabelecer os requisitos de teste. Essencialmente, esse critério requer que um conjunto de caminhos linearmente independente do grafo de programa seja executado (Pressman, 1992).

2.2.3 Técnica Baseada em Erros

Essa técnica enfatiza os erros que o programador ou o projetista pode cometer durante o processo de desenvolvimento e as abordagens que podem ser usadas para revelar a sua ocorrência. A **Análise de Mutantes** e a **Mutação de Interface** (*Interface Mutation*) são critérios típicos dessa técnica.

A Análise de Mutantes e a Mutação de Interface são critérios de teste que, para avaliar o quanto um conjunto de casos de teste **T** é adequado para o teste de um dado programa **P**, utilizam um conjunto de programas, ligeiramente diferentes de **P**, chamados de **mutantes**. O objetivo é obter casos de teste que consigam revelar as diferenças de comportamento existentes entre **P** e seus mutantes (DeMillo, 1980).

O critério Análise de Mutantes será descrito com mais detalhes por ser base para este trabalho. A síntese apresentada foi retirada do trabalho de Souza (1996).

2.2.3.1 O Critério Análise de Mutantes

A Análise de Mutantes surgiu na década de 70 na Yale University e Georgia Institute of Technology (DeMillo, 1978) e vários trabalhos empíricos e teóricos têm indicado que esse critério é atrativo para o teste de programas (Budd e DeMillo, 1980a; Mathur, 1991; Mathur e Wong, 1993; Wong, 1993; Mathur e Wong, 1994; Wong e Mathur, 1995a; Wong e Mathur, 1995b; Offut *et al.*, 1996a; Offut *et al.*, 1996b). Basicamente, a idéia da Análise de Mutantes é

[2] Medida de software que permite medir quantitativamente a complexidade lógica de um programa. Possibilita estabelecer um limite superior para o número de testes que devem ser conduzidos para garantir que todas as declarações de um programa sejam executadas pelo menos uma vez.

criar a confiança de que um programa P está correto produzindo-se, por meio de pequenas alterações sintáticas, um conjunto de programas, chamados de mutantes, semelhantes a P, e construindo-se casos de teste capazes de provocar diferenças de comportamento entre P e seus mutantes. Essas alterações são feitas utilizando-se de um conjunto de operadores denominados operadores de mutação. A cada operador associa-se um tipo ou uma classe de erros que se pretende revelar.

Segundo DeMillo (1987), a Análise de Mutantes é um critério para medir a adequação dos casos de teste, no qual um conjunto de casos de teste é dito adequado se o programa funciona corretamente quando executado com os casos de teste e se todos os programas incorretos têm um comportamento não esperado com alguns desses casos de teste. Dessa forma, o objetivo inicial da Análise de Mutantes era de servir como um meio de medir o quanto um conjunto de casos de teste é adequado a um determinado programa, sendo demonstrado por DeMillo (1978) sua eficácia também para revelar erros em programas.

A Análise de Mutantes consiste de quatro etapas principais: geração de mutantes, execução de **P** com base em um dado conjunto de casos de teste **T**, execução dos mutantes com base em **T** e análise dos mutantes. O principal objetivo é encontrar um conjunto de casos de teste que mate todos os mutantes não equivalentes a **P**. Tais conjuntos são considerados adequados para o teste de **P**, no sentido de que ou **P** está correto ou contém um erro sutil e inesperado.

Durante a Análise de Mutantes um programa **P** é testado com um conjunto de casos de teste **T**. Se o programa funciona corretamente, então este sofre pequenas modificações, gerando seus mutantes que são executados com **T**. Caso o comportamento de um mutante de **P** seja diferente de **P**, então esse mutante é dito “morto”. Caso contrário, o mutante permanece vivo devido a um dos dois motivos a seguir: 1. o conjunto **T** não é suficiente (adequado) para distinguir o comportamento de **P** e seu mutante e, com isso, novos casos de teste devem ser incluídos ao conjunto ou; 2. o mutante é dito equivalente a **P**, ou seja, para qualquer dado do domínio de entrada o comportamento de ambos os programas não difere.

Uma das dificuldades para a aplicação desse critério é a determinação de equivalência, em geral indecidível (Budd, 1981). Usualmente requer a intervenção do testador ou a elaboração de heurísticas para decidir sobre encerrar a atividade de teste com base nesse critério. Dois pontos relevantes são o projeto e a implementação de um conjunto de operadores de mutação que, essencialmente, modelam os tipos de erros que se desejam revelar. Tanto o projeto quanto a implementação baseiam-se na hipótese do programador competente que afirma que um programador competente produz programas corretos ou perto do correto. Além disso, cada

mutante é gerado pela aplicação de um único operador, considerando o efeito de acoplamento. O efeito de acoplamento afirma que casos de teste que são capazes de revelar erros simples também são capazes de revelar erros mais complexos (DeMillo, 1978).

Conseguindo-se obter casos de teste que resultem em apenas mutantes mortos e equivalentes tem-se um conjunto de casos de teste adequado ao programa em teste.

Um **escore de mutação** é definido para verificar a adequação do conjunto de casos de teste utilizado. O escore de mutação é calculado a partir da seguinte equação:

$$ms(P,T) = \frac{DM(P,T)}{M(P) - EM(P)}$$

na qual:

DM(P,T): Número de mutantes mortos pelo conjunto de casos de teste **T**.
M(P): Número de mutantes gerados para o programa **P**.
EM(P): Número de mutantes equivalentes ao programa **P**.

O escore de mutação varia no intervalo de [0,1]; quanto maior o escore mais adequado é o conjunto de casos de teste para o programa em teste. Percebe-se com essa fórmula que somente **DM(P,T)** é dependente do conjunto de casos de teste utilizado e que **EM(P)** é obtido à medida que o testador decide que um determinado mutante vivo é equivalente, o que pode ser feito interativamente ou automaticamente aplicando-se heurísticas.

A Análise de Mutantes fornece uma medida objetiva do nível de confiança na adequação dos casos de teste analisados. Com o escore de mutação, que relaciona o número de mutantes gerados com o número de mutantes mortos, pode-se avaliar a adequação dos casos de teste usados e, como conseqüência, a confiabilidade do programa testado.

Um problema para a aplicação do critério, e que por algum tempo foi um limitante para o seu uso, é o seu custo em termos de tempo de execução, pois o número de mutantes criados, mesmo para programas pequenos, pode ser muito grande, demandando um tempo de execução demasiadamente alto. Com os recursos computacionais disponíveis atualmente e com algumas estratégias torna-se possível empregar a Análise de Mutantes dentro de limites aceitáveis de tempo (Souza, 1996).

Entre as pesquisas relacionadas com os aspectos de custo de aplicação do critério Análise de Mutantes, têm sido investigadas abordagens para a redução do número de mutantes sem afetar

a eficácia do critério (Mathur e Wong, 1993; Mathur e Wong, 1994; Offut *et al.*,1996a; Offut *et al.*, 1996b). Uma linha interessante é a caracterização de um subconjunto de operadores de mutação que apresente o mesmo grau de adequação e eficácia, mas a um custo menor. Os critérios que abordam essa caracterização têm sido denominados de Mutação Restrita (*Constrained Mutation*) (Mathur, 1991; Wong, 1993), Mutação Aleatória (Acree *et al.*, 1979) e Mutação Seletiva (Offut *et al.*, 1993; Barbosa, 1998).

A **Mutação Restrita** seleciona alguns operadores específicos para a geração de mutantes. A partir desse conjunto, é possível obter uma sensível redução no custo de execução dos mutantes sem reduzir a capacidade do critério em revelar erros. A **Mutação Aleatória**, ao invés de considerar todos os mutantes gerados, seleciona aleatoriamente uma porcentagem de mutantes e a análise fica restrita a esses mutantes. A **Mutação Seletiva** propõe que os operadores de mutação responsáveis por gerar um maior número de mutantes não sejam aplicados.

O critério Análise de Mutantes para o teste de programas em C é apoiado pela ferramenta de teste *PROTEUM* desenvolvida por Delamaro (1993) e descrita na Seção 2.6.1.

2.2.3.2 O Critério Mutação de Interface

Esse critério de teste é responsável por revelar erros de integração. Possui o mesmo princípio do critério Análise de Mutantes criando mutantes inserindo pequenas perturbações nas conexões entre módulos de um programa.

Utilizando o mesmo raciocínio aplicado à Análise de Mutantes, casos de teste capazes de distinguir mutantes de interface (*interface-mutants*) também devem ser capazes de revelar grande parte dos erros de integração (Vincenzi,1998). Essa informação depende de quais mutantes são utilizados, ou seja, quais operadores de mutação são aplicados (Delamaro, 1997b).

O critério Mutação de Interface explora três idéias básicas:

- Restringir os operadores de mutação a fim de modelar apenas erros de integração.
- Testar as conexões entre módulos separadamente, uma de cada vez.
- Aplicar os operadores de mutação somente nas partes relacionadas às interfaces dos módulos, tais como chamadas de função, parâmetros ou variáveis.

O critério Mutação de Interface para o teste de integração de programas escritos em C é apoiado pela ferramenta de teste *PROTEUM/IM* desenvolvida por Delamaro (1997) e descrita na Seção 2.6.1.

2.3 A Atividade de Teste de Regressão

Estudos mostram que cerca de 2/3 do esforço total do ciclo de vida de um sistema de software é gasto durante as atividades de manutenção (Myers, 1979; Ostrand e Weyuker, 1988; Pressman, 1992; Rothermel e Harrold, 1996). Assim, o teste de software realizado após alguma modificação é tão importante quanto o teste realizado durante o processo de desenvolvimento do software.

Um software sofre modificações por várias razões, mudanças de especificação, adaptação a um novo ambiente, etc. Existem, basicamente, 4 tipos de modificações realizadas durante a manutenção (Hartmann e Robson, 1990):

- **Adaptativa**: modificações são realizadas para que o software se adapte a um novo ambiente.
- **Corretiva**: modificações são realizadas para corrigir erros.
- **Preventiva**: modificações são realizadas para facilitar futuras manutenções.
- **Evolutiva**: modificações são realizadas para adicionar novas funcionalidades.

Qualquer modificação, independentemente do tipo, pode efetivamente alterar a estrutura do programa, portanto, deve ser testada. Modificações corretivas e evolutivas requerem novos casos de teste funcionais, enquanto que, modificações adaptativas e preventivas não requerem necessariamente novos casos de teste, pois, eventualmente, não implicam mudanças nos requisitos.

As áreas modificadas devem ser tão revalidadas quanto as áreas que não sofreram modificações com o propósito de garantir que funcionalidades não modificadas não tenham sido afetadas (White e Leung, 1992). Conforme caracterizado anteriormente, o processo de teste realizado durante a fase de manutenção/evolução do software é denominada **teste de regressão**. Portanto, teste de regressão é uma parte do processo de manutenção que utiliza casos de teste previamente desenvolvidos para revelar erros nas funcionalidades não modificadas e nas novas funcionalidades, devendo ser aplicado tanto ao nível de unidade quanto aos níveis de sistema e integração. Os erros eventualmente encontrados durante a fase de manutenção são chamados de **erros de regressão** (Abdullah *et al.*, 1995).

Idealmente, quando se testa um programa que sofreu modificações, todos os casos de teste existentes devem ser reutilizados a fim de determinar se modificações introduziram

defeitos. Utilizar um conjunto de casos de teste já planejado minimiza os esforços para criar novos casos de teste e também permite uma comparação direta com a saída do programa modificado com a saída do programa original. No entanto, testar todos os conjuntos de casos de teste existentes é muito custoso. Assim, um subconjunto de casos de teste é selecionado para retestar o programa modificado. Às vezes, os casos de teste existentes não são suficientes para avaliar as modificações no software, assim, novos casos de teste devem ser adicionados.

Uma característica importante do teste de regressão é que durante os testes pode-se ter de antemão um conjunto de casos de teste disponível para reutilização. Esse conjunto permite verificar se novos erros não foram introduzidos com as modificações realizadas.

Outro ponto importante a ser ressaltado é o fato de que o teste de regressão é distinguido em duas fases (Rothermel e Harrold, 1996):

- ***Fase Preliminar***: durante essa fase os desenvolvedores corrigem o software preparando-o para a nova versão. Enquanto isso, os testadores planejam as atividades de testes ou realizam tarefas de coletar informações de *trace* e análise de cobertura. Quando as correções estão concluídas, inicia-se a próxima fase.
- ***Fase Crítica***: durante essa fase o teste de regressão é realizado. É nessa fase que os custos são maiores.

Apesar das vantagens do teste de regressão, existem basicamente dois problemas principais, os quais as técnicas existentes procuram solucionar. O primeiro - *test-update problem* - é a preocupação em manter o conjunto de casos de teste **T** ainda adequado após as modificações, pois identificar casos de teste irrelevantes e eliminá-los não é tarefa fácil. O segundo - *test-selection problem* - é a preocupação em selecionar quais casos de teste devem ser utilizados para retestar o programa após modificações. É importante que os casos de teste sejam selecionados sistematicamente, pois executar um conjunto inteiro de casos de teste adequados para validar poucas modificações pode consumir muito tempo e recursos computacionais e humanos (Hartmann e Robson, 1990).

Em geral, a atividade de teste de regressão segue os seguintes passos:

1. Identificar as modificações realizadas no programa.
2. Selecionar o conjunto de casos de teste, a partir do conjunto original T, a ser reexecutado no teste de regressão.

3. Aplicar o conjunto selecionado ao programa modificado e avaliar os resultados.
4. Gerar novos casos de teste, se necessário.
5. Aplicar os novos casos de teste e avaliar os resultados.
6. Estabelecer a base de dados dos casos de teste utilizado no teste de regressão.

Diversas técnicas são propostas na literatura para auxiliar sistematicamente o teste de regressão. Algumas dessas técnicas são descritas sucintamente a seguir.

A **Técnica Baseada em Fluxo de Dados** propõe o uso da análise de fluxo de dados para selecionar e avaliar casos de teste para serem utilizados durante o teste de regressão [OST88]. A análise de fluxo de dados pode ser utilizada para selecionar casos de teste de regressão a partir de um conjunto de casos de teste já definido, e gerar casos de teste adicionais. A análise de fluxo de dados permite selecionar casos de teste para todas essas situações. A teoria do teste de regressão utilizando a análise de fluxo de dados é executar todos os caminhos que sofreram modificações. Na mesma linha de pesquisa baseada em fluxo de dados, a **Técnica baseada na Cobertura dos Critérios Potenciais-Usos**, proposta por Granja (1997), seleciona os casos de teste para serem utilizados no teste de regressão a partir dos elementos requeridos para os critérios Potenciais-Usos que foram modificados.

A **Técnica Baseada em Execução Simbólica,** proposta por Yau e Kishimoto (1987), divide o domínio de entrada em diferentes classes utilizando o código e a especificação do programa modificado e seleciona um caso de teste de cada classe para executar o programa modificado. O objetivo é executar cada modificação ou cada novo código pelo menos um vez. O particionamento do domínio é baseado na especificação a fim de obter diferentes combinações dos dados de entrada e a identificação de possíveis caminhos através do código, estabelecendo que cada classe de entrada deve ser testada pelo menos uma vez. Assim, casos de teste são alocados a diferentes partições de entrada e executados utilizando execução simbólica.

A **Técnica Baseada em Domínio**, proposta por Mayrhauser *et al.* (1994), gera casos de teste baseando-se na análise e no modelo do domínio. Os modelos de domínio são utilizados como uma estrutura para gerar casos de teste e armazenam informações sintáticas e semânticas necessárias para a geração de novos casos de teste.

A **Técnica baseada em Cobertura**, proposta por Rothermel e Harrold (1997), seleciona casos de teste de regressão percorrendo os grafos de fluxo de controle dos programas originais e modificados. Basicamente, o algoritmo constrói o grafo de fluxo de controle para os programas P e P´ e coletando *traces* de execução que associam casos de teste em T com arcos dos grafos.

Durante o percurso, o algoritmo compara as declarações associadas com os nós que são simultaneamente alcançados em ambos os grafos. Quando o algoritmo encontra um par de nós em N e N´ dos grafos de P e P´, respectivamente, tal que as declarações associadas a N e N´ não sejam lexicamente idênticas, o algoritmo seleciona todos os casos de teste em T que alcançam N em P.

Diante da diversidade de técnicas de teste de regressão existentes, são necessários mecanismos que possibilitem avaliar e comparar essas técnicas. Rothermel e Harrold (1996) propuseram um *framework* para avaliar e comparar técnicas baseadas na abordagem seletiva. Além disso, Leung e White (1991) propõem um modelo de custo para comparar as técnicas baseadas na abordagem seletiva e as técnicas baseadas na abordagem *retest-all*. Nas seções 2.3.1 e 2.3.2 esses mecanismos são apresentados, respectivamente.

Nas seções 2.4 e 2.5 são apresentadas com mais detalhes as técnicas que foram utilizadas no escopo deste trabalho: **Técnica baseada em Modificação** e **Técnica baseada em Mutação Seletiva**.

2.3.1 *Framework* de Avaliação e Comparação de Técnicas de Teste de Regressão Seletivas

Algumas técnicas de teste de regressão baseiam-se na especificação do software para selecionar os testes de regressão, enquanto outras se baseiam no código original (**P**) e no código modificado (**P'**). Essas últimas técnicas têm os seguintes objetivos (Rothermel e Harrold, 1996; Souza, 1997):

- **Cobertura** - Selecionar casos de teste que passem pelos componentes modificados.
- **Minimização** - Semelhante à anterior: selecionar um conjunto minimizado de casos de teste de regressão que passem pelos componentes modificados.
- **Segurança** - Selecionar casos de teste de **T** que revelam defeitos em **P'**.

Como são muitas as técnicas seletivas de teste de regressão, é preciso uma maneira de compará-las e avaliá-las. Com esse objetivo, Rothermel e Harrold (1996) propuseram um *framework* que define algumas características que auxiliam na comparação e avaliação das técnicas: inclusão (*inclusiveness*), precisão, eficiência, e generalidade.

- **Inclusão** - Mede a extensão com a qual a técnica inclui casos de teste que fazem com que **P'** produza uma saída diferente de **P** e revela os defeitos.
- **Precisão** - Mede a habilidade da técnica em evitar testes que não fazem com que **P'** produza um saída diferente de **P**.
- **Eficiência** - Mede o custo computacional da técnica.
- **Generalidade** - Mede a habilidade da técnica em manipular linguagens diversas.

Segundo Rothermel e Harrold, uma técnica seletiva de teste típica segue o seguinte:

1. Selecionar um conjunto de casos de teste **T'**$\subseteq$ **T** para executar **P'**.
2. Testar **P'** com **T'** para estabelecer a corretitude de **P'** com respeito a **T'**.
3. Se necessário, criar **T"**, um conjunto de novos casos de teste para **P'**.
4. Testar **P'** com **T"** para estabelecer a corretitude de **P'** com respeito a **T"**.

Para a definição do *framework*, os autores dividem os casos de teste em três classes: *fault-revealing, modification-revealing* e *modification-traversing*.

Fault-revealing são os casos de teste capazes de revelar os erros existentes no programa. Entretanto, não existe um mecanismo eficiente para e ncontrar esses casos de teste a partir de um conjunto inicial. Assim, tem-se a seguinte indagação de difícil reposta: **Como saber se um caso de teste revela erros antes de aplicá-lo**?

Modification-revealing são os casos de teste que revelam um comportamento diferente entre **P** e **P'**. Entretanto, para selecioná-los seria necessário executar todo o conjunto de casos de teste para saber quais são *modification-revealing*. Então, a solução é selecionar aqueles casos de teste que passam pelas modificações feitas no programa: os casos de teste *modification-traversing*, que incluem os casos de teste *fault-revealing* e *modification*-revealing. Assim, os casos de teste *modification-traversing* são aqueles com maiores chances de revelarem defeitos. A Figura 2.1 mostra a relação entre as três diferentes classes de conjuntos de casos de teste.

Além das três classes de conjuntos de casos de teste definidos para este *framework*, têm-se os casos de teste obsoletos e não obsoletos. Os casos de teste obsoletos são aqueles que não serão reutilizados para as atividades de teste de regressão, enquanto que os casos de teste não obsoletos são aqueles que serão reutilizados. Ambos os casos de teste estão caracterizados na Figura 2.1.

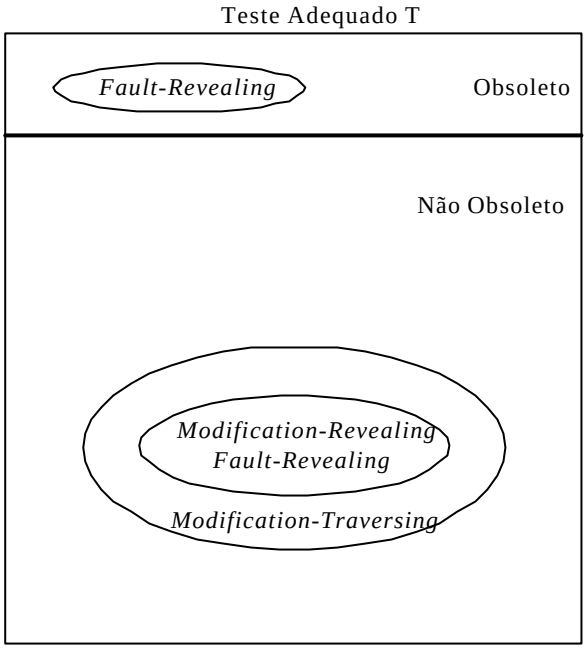

Figura 2.1 - Relacionamento entre as três classes de conjuntos de casos de teste (Rothermel e Harrold, 1996).

As três classes de casos de teste descritas podem contribuir para a especificação do *framework* para avaliar e comparar técnicas de teste de regressão baseadas na abordagem seletiva por várias razões:

- Testadores são relutantes quanto ao fato de descartar casos de teste que possam revelar erros. O relacionamento entre as três classes de casos de teste fornece uma maneira de avaliar analiticamente as técnicas de teste baseadas na abordagem seletiva em termos de suas habilidades de selecionar casos de teste que sejam *fault-revealing* e casos de teste que não o sejam. Apesar do fato de que essas técnicas objetivam selecionar casos de teste para satisfazer alguma medida de adequação, é razoável e importante avaliar essas técnicas em termos de suas habilidades de revelar defeitos.
- As três classes podem ainda servir para distinguir técnicas de teste de regressão baseadas na abordagem seletiva. Muitas técnicas são mais precisas, eliminando do conjunto de casos de teste aqueles não que executam componentes modificados e aqueles que não causam saídas diferentes em **P** e **P'**. Assim, é útil comparar técnicas de teste de regressão em termos de suas habilidades em identificar tais classes de casos de teste.

A seguir, cada uma das quatro características que compõem o *framework* é apresentada. A seguinte notação é utilizada: **M** = técnica; **P** = programa original; **P'** = programa modificado;

T = conjunto de casos de teste original; e **T'** = subconjunto de casos de teste de regressão, **T'** $\subseteq$ **T**.

Inclusão

Mede a extensão com a qual a técnica **M** seleciona casos de teste *modification-revealing* a partir de **T** para incluir em **T'**. Assim, tem-se:

Definição 1 - Supondo que **T** contenha **n** casos de teste que sejam *modification-revealing* para **P** e **P'**, e supondo que **M** selecione **m** desses casos de teste, a inclusão de **M** relativa a **P**, **P'** e **T** é:

(1) a porcentagem dada pela expressão (100(m/n)) se $n \neq 0$ ou

(2) 100% se n=0.

Por exemplo, se **T** contém 50 casos de teste dos quais 8 são *modification-revealing* para **P** e **P'**, e **M** seleciona 2 destes 8, então a inclusão de **M** relativa a **P**, **P'** e **T** é 25%.

Definição 2 - Se para todo **P**, **P'** e **T**, a inclusão de **M** relativa a **P**, **P'** e **T** é 100%, então **M** é segura.

Inclusão e segurança são medidas importantes e bastante significantes. Se uma técnica de teste de regressão **M1** tem maior inclusão que uma técnica **M2**, então **M1** tem maior habilidade em revelar defeitos do que **M2**.

Precisão

Mede a extensão com a qual **M** omite casos de teste que não são *modification-revealing*. Assim, tem-se:

Definição 3 - Supondo que **T** contenha **n** casos de teste que não são *modification-revealing* para **P**, **P'** e **T** e supondo que **M** omite **m** destes casos de teste. A precisão de **M** relativa a **P**, **P'** e **T** é:

(1) a porcentagem dada pela expressão (100(m/n)) se $n \neq 0$ ou

(2) 100% se n=0.

Por exemplo, se **T** contém 50 casos de teste dos quais 44 não são *modification-revealing* para **P**, **P'** e **T**, e **M** omita 33 desses 44, então a precisão de **M** relativa a **P**, **P'** e **T** é 75%.

A precisão é bastante útil, pois mede a extensão com a qual **M** evita selecionar testes que não causam um comportamento diferente no programa modificado. Em geral, quando se compara técnicas de teste em termos de precisão, é possível identificarem-se as técnicas que menos escolhem casos de teste desnecessários. Quando se compara segurança em termos de precisão, pode-se identificar técnicas que se aproximam do objetivo principal que é selecionar casos de teste que sejam *modification-revealing*.

Eficiência

A eficiência de uma técnica é medida em termos do espaço utilizado para armazenar os dados de teste e em termos do tempo gasto para aplicar e avaliar a técnica. Quando tempo é considerado, a estratégia seletiva é mais econômica que a estratégia *retest-all* se o custo de selecionar **T'** é menor que o custo de executar os casos de teste em **T** (Leung e White, 1991).

Podem-se identificar fatores importantes que influenciam na avaliação da eficiência de uma técnica:

- Fases que são realizadas durante o teste de regressão: fase preliminar e fase crítica.
- Esforço humano empregado durante o teste de regressão. Técnicas que requerem muito esforço humano são impraticáveis.
- Extensão com a qual a técnica deve calcular as informações nos módulos modificados. Uma técnica que deve determinar cada componente removido, modificado ou adicionado a **P** ou calcular semelhanças entre **P** e **P'** pode ser mais cara que aquela técnica que verifica essas modificações apenas quando necessário.
- Habilidade da técnica em manipular casos nos quais **P'** é criado a partir de várias modificações de **P**. Uma técnica que depende da análise de programas e processa uma modificação de cada vez pode ser forçada a reavaliar todas as informações obtidas após cada modificação. Tal análise pode ser cara e pode aumentar o custo de aplicação da técnica.

Generalidade

A generalidade de uma técnica seletiva é a habilidade da técnica em funcionar em diversas situações. Fatores que devem ser considerados ao avaliar a generalidade de uma técnica:

- Para ser prática, a técnica deve funcionar para vários tipos de programas. Por exemplo, uma técnica que é definida somente para procedimentos com **if**, **while**, e atribuições não é prática.
- Uma técnica deve manipular modificações realistas. Por exemplo, uma técnica que não manipula modificações que alteram o fluxo de controle, em geral, não é prática.
- Uma técnica que depende do ambiente de teste e de manutenção é menos geral que uma que não depende desses fatores.
- Uma técnica que depende da disponibilidade de ferramentas de análise é menos geral que uma técnica que não depende de tais ferramentas. Por exemplo, uma técnica que requer informações sobre *trace* de casos de teste é menos geral que uma técnica que não requer essas informações.
- Uma técnica pode apoiar teste intraprocedural ou interprocedural. Na prática, teste de regressão é freqüentemente realizado em nível interprocedural.

Pode-se definir generalidade mais quantitativamente. No entanto, neste *framework*, as comparações qualitativas são suficientes, segundo Rothermel e Harrold (1996).

Com o objetivo de utilizar e avaliar o *framework* definido, Rothermel e Harrold conduziram um experimento aplicando várias técnicas de teste de regressão seletivas em fragmentos de programas modificados. Cada uma das técnicas foi aplicada e avaliada de acordo com as características definidas no *framework*. A partir desse experimento, Rothermel e Harrold estabeleceram os seguintes pontos:

- Aumentar a precisão pode diminuir a eficiência em técnicas seguras e não seguras.
- Aumentar a inclusão pode diminuir a eficiência em técnicas não seguras.
- Os fatores que afetam a generalidade também podem afetar a inclusão, a precisão e a eficiência.

- Uma técnica que deve reavaliar todas as informações já obtidas a cada modificação pode influenciar a eficiência da técnica. Por outro lado, se a reavaliação não ocorre a precisão da técnica pode ser influenciada.

2.3.2 Um Modelo de Custo para Comparar Técnicas Seletivas e Técnicas *Retest-all*

Em princípio, a abordagem seletiva requer mais tempo e recursos para selecionar casos de teste a fim de reduzir o número de casos de teste a serem executados. Um benefício é alcançado somente se o esforço gasto na seleção for menor que o esforço de executar todos os casos de teste originais, como faz a abordagem *retest-all*.

Muitos fatores influenciam o custo do teste de regressão. Esses fatores podem ser de dois tipos: direto e indireto. **Direto**: inclui custos com as atividades relacionadas ao teste e aos recursos físicos utilizados para execução dos casos de teste. **Indireto**: inclui custos com gerenciamento do processo de teste, base de dados para armazenar informações sobre os casos de teste, os resultados da análise estática e históricos de execução. No modelo de custo, proposto por Leung e White (1991), somente os custos diretos são considerados.

O custo de aplicar um conjunto de casos de teste a um sistema consiste dos seguintes componentes:

- **Custo de Análise do Sistema** (**Ca**) - Envolve custos gastos com o estudo da especificação, do projeto e do código do sistema. Esse estudo deve facilitar no julgamento do comportamento do sistema quando da execução dos casos de teste de regressão.
- **Custo de Seleção de Teste** (**Cs**) - Envolve custos gastos em selecionar os casos de teste para testar o comportamento atual do sistema. Esse custo depende muito da abordagem escolhida e também da técnica.
- **Custo de Execução de Teste** (**Ce**) - Envolve custos gastos em preparar o ambiente para o teste e na execução propriamente dita.
- **Custo de Análise dos Resultados** (**Cr**) - Envolve custos gastos em checar o comportamento do sistema sob a execução dos casos de teste. Vários fatores influenciam esses custos: tempo para coletar os resultados de saída, tempo para comparar o resultado obtido com o resultado esperado, tempo para registrar esses dados.

Esses componentes dependem de vários fatores e alguns deles estão interrelacionados:

- A abordagem de teste utilizada influencia muito o custo. Ex.: a abordagem que utiliza a técnica de caixa-preta (funcional), que requer apenas a análise da especificação, tem um custo menor que a abordagem que utiliza ambas as técnicas caixa-preta e caixa-branca (estrutural), que requer além da análise da especificação a análise do código fonte.
- O número de casos de teste também depende da técnica de teste. Ex.: uma técnica que requer que todos os pares de **definição-uso** sejam executados geralmente necessita de mais casos de teste do que aquela técnica que requer somente que todas as instruções sejam executadas. O número de casos de teste influenciam nos valores de **Cs**, **Ce** e **Cr**.
- A complexidade do sistema influencia muito o custo. Quanto maior a complexidade maior é o número de casos de teste e maior é o tempo gasto com a análise do sistema (**Ca**).

Basicamente, custo de aplicação de uma abordagem **E** com relação a um conjunto de casos de teste **T** é definido da seguinte forma:

$$\boldsymbol{E} = Ca(T) + Cs(T) + Ce(T) + Cr(T)$$

na qual, **Cs**, **Ce** e **Cr** são dependentes do número de casos de teste.

Algumas considerações foram feitas para comparar os custos das abordagens *retest-all* e *seletiva*:

- Ambas as abordagens são igualmente eficazes.
- A abordagem *retest-all* não realiza análises antes de aplicar todos os casos de teste.
- A abordagem seletiva gasta muito esforço em selecionar um subconjunto de casos de teste para execução.

Segundo Leung e White (1991), o modelo de custo apresentado é apenas um passo inicial para modelar os esforços gastos nas atividades de teste de regressão. O custo pode ainda ser refinado para incorporar fatores indiretos tais como: gerenciamento, custo para armazenar as

informações relacionadas aos casos de teste e custos com desenvolvimento de ferramentas para auxiliar as atividades de teste de regressão. A inclusão desses fatores pode não afetar os resultados já encontrados. No entanto, esses fatores podem ser muito importantes para avaliar a qualidade das abordagens utilizadas para os testes de regressão. Estudos preliminares indicam que quanto maior a complexidade, maior é o número de casos de teste a serem executados, significando assim, um maior tempo gasto. Assim, a abordagem seletiva é mais atrativa que a abordagem *retest-all* se os custos com gerenciamento forem considerados.

Considerando custos com armazenamento, a abordagem *retest-all* torna-se mais atrativa que a abordagem seletiva, pois esta abordagem armazena mais dados, tais como: informações da análise estática e dos históricos de cada caso de teste. Essas informações podem dobrar o tamanho da base de dados utilizada para o teste de regressão. O custo com desenvolvimento de ferramentas não acarreta maior impacto aos resultados obtidos. Esse custo representa um fator constante para ambas as abordagens.

Leung e White chegaram às seguintes conclusões definindo o modelo de custo descrito:

- O custo da aplicação de uma abordagem de teste depende de, basicamente, 4 fatores: seleção dos conjuntos de casos de teste de regressão, execução dos conjuntos de casos de teste de regressão, análise dos resultados e razão entre o tamanho do conjunto selecionado com o tamanho do conjunto de casos de teste original.
- O custo da aplicação de ambas as abordagens depende muito da técnica, da complexidade do programa em teste e do ambiente utilizados.
- Existe uma relação linear entre os valores de **Cs**, **Ce**, **Cr** e o número de casos de teste utilizado.
- Duas áreas de estudo promissoras são o relacionamento entre a confiabilidade dos produtos de software e as abordagens de teste, e o relacionamento entre o custo da atividade de teste e a obtenção da qualidade desejada. Apesar de se desejar uma alta confiabilidade, é igualmente importante relacionar a confiabilidade a um custo razoável.

Tanto o modelo de custo quanto o *framework* de avaliação fornecem uma sistemática clara de se avaliar e comparar abordagens e técnicas de teste de regressão. Ambos podem ser utilizados de forma complementar a fim de melhorar a qualidade de se avaliar o teste de regressão.

2.4 Técnica de Teste de Regressão baseada em Modificação

O principal objetivo desta técnica é selecionar aqueles casos de teste que revelam um comportamento diferente entre o programa modificado e o programa original. Assim, dado um programa **P**, seu conjunto de casos de teste de regressão **T** e seu programa modificado **P'**, é preciso encontrar **T'** $\subseteq$ **T** tal que:

$$\forall t \in T, t \in T' \Leftrightarrow P'(t) \neq P(t)$$

No entanto, encontrar **T'** não é uma tarefa fácil. Na prática, segundo Wong *et al.* (1997a), é possível encontrar **T'** somente executando **P'** com cada caso de teste de regressão em **T**. Mas essa tarefa não é viável, pois é muito custosa. Entretanto, ao invés de encontrar-se **T'** a partir de **T**, pode-se encontrar **T"**, **T'** $\subseteq$ **T"**, incluindo todos os casos de teste em **T** que executam o código modificado, ou seja, os casos de teste *modification-traversing*. Assim:

$\forall t \in T, t \in T" \Leftrightarrow$ *t executa código em P que foi modificado ou removido para gerar P' ou código em P no qual novo código foi adicionado para gerar P'*

Portanto, **T"** é um subconjunto de **T** baseado em modificação. A Figura 2.2 ilustra o esquema apresentado. Apesar do conjunto **T"** ser possuir todos os casos de teste que passam por alguma modificação, ele pode conter, ainda, casos de teste que não revelam um comportamento diferente entre **P** e **P'**, sendo assim, não preciso. Assim, a seguinte questão: "**Que procedimentos ou mecanismos podem ser aplicados para reduzir o número de casos de teste visando a eliminar aqueles casos de teste que não revelam comportamento diferente**?" Os mecanismos de minimização e priorização são propostos para solucionar essa questão.

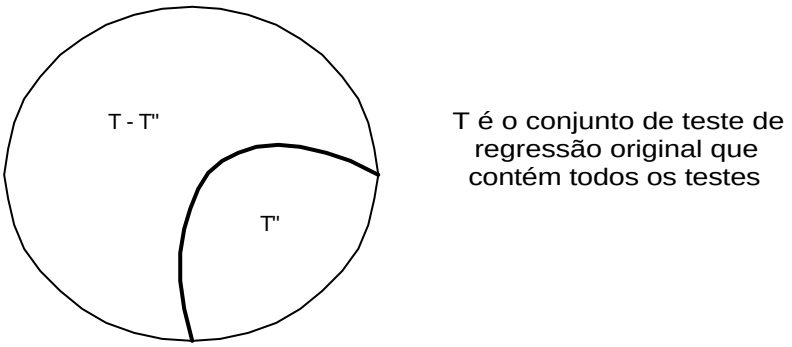

Figura 2.2 - Conjunto T" baseado em modificação.

O **mecanismo de minimização** encontra um subconjunto minimizado de casos de teste de regressão que preserva a cobertura com respeito a um dado critério a partir de **T**. O **mecanismo de prioridades** seleciona casos de teste de regressão utilizando o mecanismo de minimização de acordo com a análise de cobertura de cada caso de teste segundo um dado critério. Sob esse cenário, testadores podem selecionar os casos de teste no topo da lista de prioridades. Tal flexibilidade não aparece em nenhuma outra técnica de revalidação seletiva. O uso de ambos os mecanismos oferece algumas vantagens:

- Reduz a quantidade de esforço requerido para testar o conjunto minimizado.
- Aumentam as chances de se escolherem casos de teste que produzam saídas diferentes na antiga e na nova versão do programa.
- Diminuem as chances de se incluírem casos de teste que não diferenciem o novo e o antigo programa, ou seja, não releve as possíveis falhas existentes.

Para selecionar o conjunto de casos de teste de regressão mínimo e priorizado a partir de um conjunto **T** adequado, 2 passos são realizados:

- Construção de um superconjunto de todos os casos de teste de regressão que devem ser utilizados.
- Se necessário, utilizar a minimização e priorização.

Esses dois passos provêem um guia sobre quais casos de teste devem ser utilizados para executar as modificações realizadas. A aplicação desses dois passos está ilustrada na Figura 2.3 a seguir.

Primeiramente, os casos de teste *modification-traversing*, conjunto T", são selecionados a partir do conjunto de casos de teste de regressão T disponível e das modificações realizadas. Em seguida, se necessário, os mecanismos de minimização e priorização são aplicados. Cada mecanismo gera um subconjunto de teste de regressão diferente a partir do conjunto T". Note que esses mecanismos não são aplicados seguidos um do outro, e sim de forma separada. No entanto, um mecanismo poderia ser utilizado como complemento do outro a fim de reduzir ainda

mais o número de casos de teste a serem reexecutados. Após esse processo, caso seja necessário, novos casos de teste são gerados.

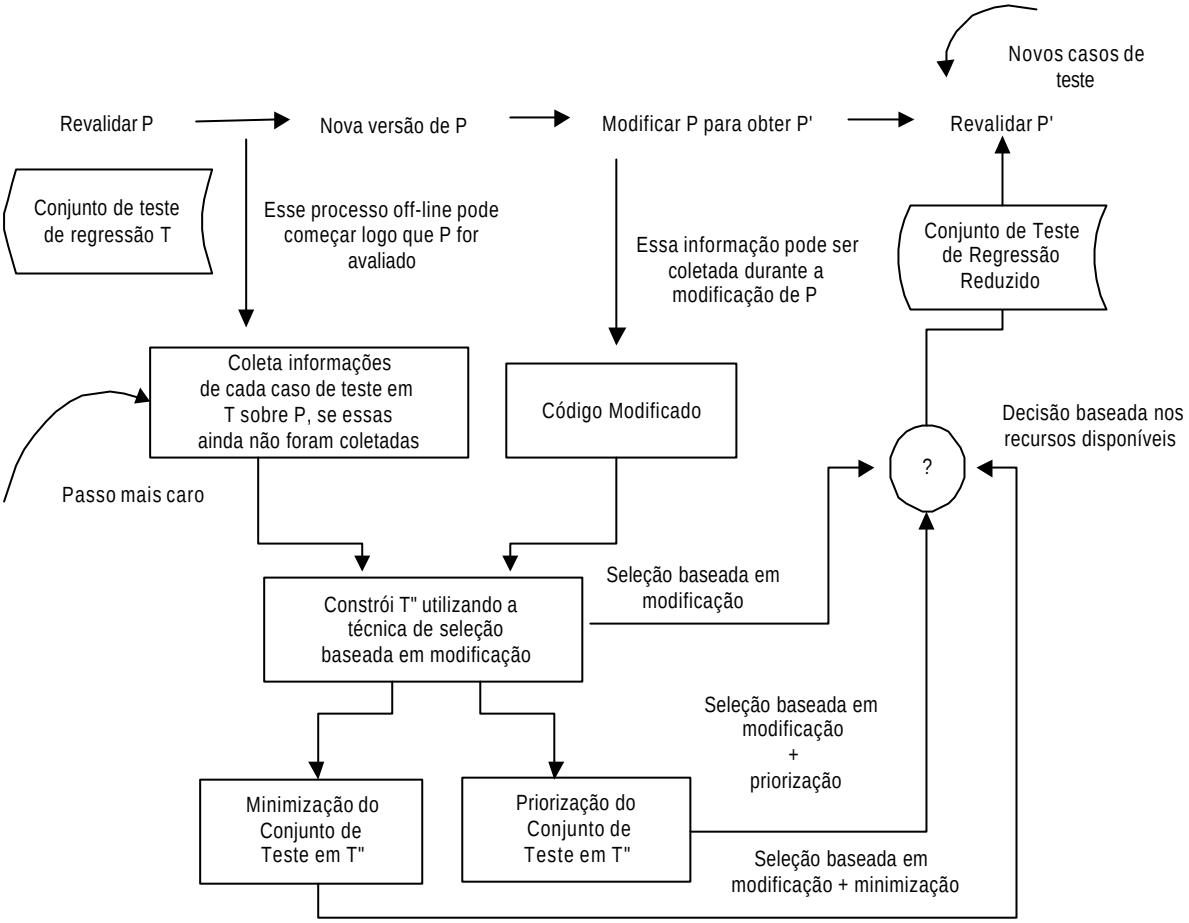

Figura 2.3 - Seleção dos casos de teste de regressão utilizando um processo off-line (Wong et al.,1997a).

A fim de avaliar a técnica baseada em Modificação, Wong *et al.* conduziram um experimento utilizando o programa SPACE[3], desenvolvido pela Agência Espacial Européia, e a ferramenta de teste *ATAC* (descrita na Seção 2.6.1). Para realização desse estudo de caso, obteve-se uma lista de erros, Tabela 2.1, que foi elaborada durante as fases de teste e integração do SPACE. Com base nessa lista de erros, dez programas com defeitos foram gerados. Para cada um desses programas com defeitos, conjuntos de casos de teste de regressão baseados em modificação foram gerados, e a partir desses conjuntos baseados em modificação, conjuntos minimizados e priorizados foram selecionados seguindo o processo ilustrado na Figura 2.3. Os

[3] O programa SPACE permite que se possa descrever a configuração de um vetor de antenas utilizando-se de uma linguagem de alto nível.

conjuntos de casos de teste de regressão baseados em modificação foram selecionados a partir de um conjunto original de 1000 casos de teste.

Tabela 2.1 – Lista de erros utilizados para gerar os programas com defeitos.

Equações incorretas ou insuficientes
Alterações nos comandos de condições
Dados acessados ou armazenados incorretamente
Variáveis incorretas

O mecanismo de minimização utilizado no experimento foi implementado na ferramenta ATAC, descrita na seção 2.6 deste capítulo. Já o mecanismo de priorização utilizou o incremento de cobertura dos critérios de teste todos-usos, blocos e decisões para selecionar os conjuntos priorizados. Casos de teste *modification-traversing* foram sendo inseridos no topo da lista de prioridades à medida que esses casos melhoravam a cobertura com relação a cada um dos critérios. Utilizando-se dessa lista de prioridades, os conjuntos priorizados para cada defeito foram definidos com os **n** primeiros casos de teste da lista. O valor de **n**, segundo Wong *et al.* é desconhecido, pois não se sabe quantos casos de teste que estão na lista de prioridades devem ser selecionados. Assim, no experimento conduzido por Wong *et al.* foi definido que o valor máximo de **n** era o número de casos de teste selecionados pelo mecanismo de minimização Foram selecionados três conjuntos priorizados diferentes para cada defeito: um conjunto com 1/3 do número de casos de teste dos conjuntos minimizados, outro com 2/3 desse número e outro com 3/3 desse número.

A partir desse experimento realizado, Wong *et al.* concluíram:

- Um teste de regressão é um *trade-off* entre o número de casos de teste de regressão necessários e seu custo.
- Quanto maior o número de casos de teste de regressão, mais completo é o processo de revalidação, no entanto, esse processo requer muitos recursos e, muitas vezes, pode não ser prático.
- Executar poucos casos de teste pode ser mais barato, mas pode comprometer a garantia de que as novas funcionalidades sejam validadas e verificadas, ou seja, pode comprometer a eficácia e a eficiência do teste de regressão.

Assim, a utilização da técnica baseada em modificação, juntamente com a minimização e priorização, permite a escolha de um conjunto eficaz para realização dos teste de regressão com um menor custo.

2.5 Técnica de Teste de Regressão baseada em Mutação Seletiva

Uma característica especial da seleção de casos de teste de regressão utilizando mutação seletiva é que, como mostra a Figura 3.4, a seleção pode ser inicializada logo que a nova versão do programa é finalizada. A solução apresentada por esta técnica depende somente do programa original, do conjunto dos casos de teste de regressão e do conjunto de operadores de mutação. Ou seja, a seleção dos casos de teste pode ser realizada por um processo *off-line* antes que qualquer modificação seja realizada.

A principal estratégia desta técnica é examinar como um critério **C** pode ser utilizado no teste de regressão a fim de auxiliar testadores a determinarem quais casos de teste devem ser selecionados ou têm maior prioridade para revalidar as funcionalidades herdadas da versão anterior do programa, e quais devem ser omitidos ou têm baixa prioridade para tal revalidação. A qualidade de **C** e dos casos de teste selecionados pode ser medido com o auxílio de duas métricas:

- Redução do número de casos de teste a serem reexecutados.
- Número de defeitos que podem ser revelados.

A mutação seletiva, como já mencionado, consiste em selecionar um conjunto de operadores de mutação para encontrar o conjunto minimizado de casos de teste a serem aplicados durante os testes de regressão. Estudos empíricos demonstram que a mutação seletiva é bastante eficiente em revelar defeitos e pode contribuir para a redução dos custos aplicados nas atividades de teste (Mathur e Wong, 1994; Offut *et al.*, 1996a, Offut *et al.*, 1996b, Wong e Mathur, 1995a; Wong e Mathur, 1995b; Wong *et al.*, 1997c).

A técnica é aplicada da seguinte maneira (Figura 2.4). Inicialmente, o conjunto de casos de teste T disponível é chamado de conjunto de casos de teste de regressão T e o conjunto de teste de regressão reduzido T’ está vazio. Utilizando-se de um conjunto de operadores de mutação selecionado a partir da mutação seletiva e do conjunto de casos de teste de regressão T, o conjunto de casos de teste T’ é definido. Se um caso de teste do conjunto T melhora o escore

de mutação com relação à mutação seletiva, esse caso de teste é inserido no conjunto de teste de regressão reduzido T'. Esse processo continua até que todos os casos de teste do conjunto T sejam examinados. Ao final do processo, o conjunto T' é selecionado. Se necessário, novos casos de teste são gerados.

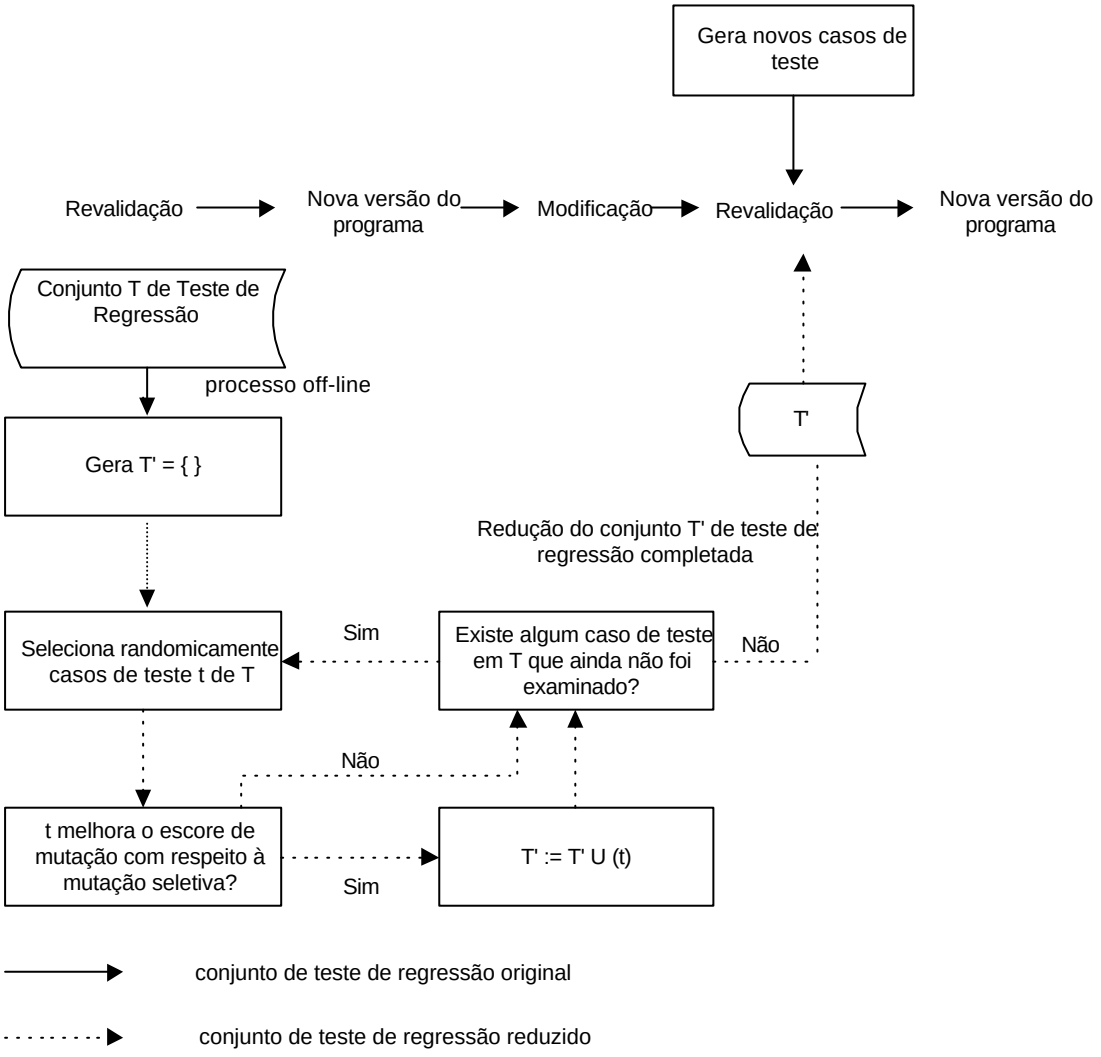

Figura 2.4 - Redução do conjunto de teste de regressão (Wong et al. 1997b).

Segundo Wong *et al.* (1997b), o custo do teste de regressão pode ser medido de várias maneiras: tempo necessário para executar os casos de teste e tempo despendido pelo testador em analisar os resultados obtidos. Ambos os custos são proporcionais ao número de casos de teste a serem reexecutados.

Para avaliar a aplicação da técnica baseada em Mutação Seletiva, Wong *et al.* conduziram um experimento utilizando um conjunto de programas utilitários do UNIX. Os seguintes passos foram realizados para avaliar a técnica:

Passo 1 - Preparação dos dados dos programas

Os programas utilizados foram nove utilitários do UNIX, vide Tabela 2.2. Para cada um desses programas, uma lista de erros e um conjunto de casos de teste de regressão foram estabelecidos. A Tabela 2.3 apresenta a lista de erros utilizada. Baseando-se nessa lista de erros, os programas com defeitos foram gerados.

Tabela 2.2 - Programas utilitários do UNIX utilizados nos experimentos.

Programas	Descrição
Cal	Imprime calendário de um ano ou mês específico
Checkeq	Reporta delimitadores desbalanceados
Col	Filtra arquivos tratando caracteres especiais
Comm	Seleciona ou rejeita linhas comuns de dois arquivos classificados
Crypt	Criptografa ou decriptografa arquivos utilizando *password*
Look	Encontra palavras em um sistema ou linhas
Sort	Classifica e realiza merge em arquivos
Spline	Interpola pontos
Tr	Traduz caracteres
Uniq	Reporta ou remove linhas adjacentes duplicadas

Tabela 2.3 - Tipos de defeitos utilizados nos experimentos.

Caminhos removidos	
Predicados incorretos	Troca de operador relacional
	Troca de operador lógico
Declarações incorretas	Inicialização incorreta
	Constante incorreta
	Precedência incorreta
	Referência de vetor incorreta
	Operação de ponteiro incorreta
	Troca de mesmo tipo de variável
	Troca de operador aritmético
	Miscelânea
Declarações removidas	Remoção completa de declaração
	Remoção parcial de declaração
Número incorreto de iterações	
Predicados removidos	

Passo 2 - Seleção dos Operadores de Mutação

Doze operadores de mutação da ferramenta *PROTEUM*, descrita na Seção 2.6 deste capítulo, foram selecionados, de acordo com estudos realizados por (Offut *et al.*, 1996b; Wong e Mathur, 1995a; Wong e Mathur, 1995b; Wong *et al.*, 1997c), para que os mutantes fossem gerados para cada programa. A Tabela 2.4 mostra os operadores de mutação utilizados para cada critério de mutação seletiva. O Apêndice A apresenta detalhes sobre cada um desses operadores de mutação.

Mutantes estruturais e declarativos foram gerados e avaliados. Mutantes estruturais modificam a estrutura do programa e estão relacionados com fluxo de controle e/ou fluxo de dados de algumas variáveis do programa. Mutantes declarativos realizam pequenas mudanças sintáticas nas declarações sem alterar a estrutura.

Tabela 2.4 - Operadores de mutação e critérios de mutação seletiva.

Critério	Operador de Mutação	Nível
MUT-A	VDTR, VTWD	Declarativo
MUT-B	SSDL	Estrutural
MUT-C	SMVB, SBRn, SCRn	Estrutural
MUT-D	Vprr, Vtrr, CRCR	Declarativo
MUT-E	OLLN, OLNG, ORRN	Estrutural

Passo 3 – Seleção dos Conjuntos de Casos de Teste de Regressão Reduzidos

O esquema da Figura 2.4 ilustra como foi realizada a seleção dos conjuntos de casos de teste de regressão reduzido. A partir de um conjunto de casos de teste de regressão **T**, selecionaram-se **t** casos de teste aleatoriamente, **t** ∈ **T**, gerando um conjunto reduzido de casos de teste de regressão **T'**. Se o caso de teste **t** melhorasse o escore de mutação com relação à mutação seletiva, esse era incluído no conjunto **T'**. Esse processo foi realizado até que todos os casos de teste foram avaliados. Uma vez incluído no conjunto **T'**, o caso de teste não é excluído devido à inclusão de qualquer outro caso de teste.

Para cada programa (Tabela 2.2) foram gerados cinco conjuntos de casos de teste (T_n) a partir do conjunto de casos de teste original T em diferentes ordens. Para cada um desses conjuntos foram selecionados cinco subconjuntos de casos de teste (T'_n) para serem utilizados no teste de regressão. O primeiro conjunto, T_0, é composto por todos casos de teste disponíveis ordenados de forma crescente, enquanto que os demais conjuntos, T_1, T_2, T_3 e T_4 são permutações do conjunto T_0 obtidas aleatoriamente. Cada caso de teste recebe uma numeração

para ser executado pela ferramenta *PROTEUM.* Esse é a numeração que foi permutada para gerar os conjuntos T_1, T_2, T_3 e T_4.

A técnica baseada em mutação seletiva mostrou-se bastante promissora, pois permitiu uma grande redução dos números de casos de teste a serem reexecutados sem comprometer a eficácia dos conjuntos em relevar os defeitos de cada programa utilitário do UNIX.

2.6 Ferramentas de Teste e Estudos Empíricos

As atividades de teste de software são bastante custosas, demandando muitos esforços de desenvolvedores, programadores e testadores, como já mencionado. Assim, com o intuito de reduzir os custos aplicados nessas atividades, diversas ferramentas de teste foram desenvolvidas. Com o auxílio dessas ferramentas, diversos estudos empíricos também têm sido realizados visando a estabelecer estratégias e critérios que auxiliem na redução desses esforços. Muitos desses estudos são relevantes para este trabalho, portanto, são sintetizados nessa seção juntamente com algumas ferramentas de teste de software.

2.6.1 Ferramentas de Teste

A seguir uma síntese de algumas ferramentas de teste é apresentada. As ferramentas de teste *PROTEUM, PROTEUM/IM,* que apóiam as atividades de teste durante o desenvolvimento do software, são descritas com mais detalhes, pois foram utilizados nos experimentos descritos no Capítulo 3. Outras ferramentas, como: *ATAC, POKETOOL, RePOKETOOL, DEJAVU* são descritas mais sucintamente.

A ferramenta *PROTEUM*

A *PROTEUM* (Delamaro, 1993) apóia o critério Análise de Mutantes e está configurada para funcionar em estações SUN sob o ambiente *OPENWINDOWS.* A ferramenta apresenta interface gráfica e oferece recursos para a execução das seguintes operações: definição de casos de teste, execução do programa em teste, seleção dos operadores de mutação que serão utilizados para gerar os mutantes, geração dos mutantes, execução dos mutantes com os casos de teste definidos, análise dos mutantes vivos e cálculo do escore de mutação. As funções implementadas na *PROTEUM* possibilitam que alguns desses recursos sejam executados automaticamente, enquanto que para outros são fornecidas facilidades para que o testador possa realizá-los. A

PROTEUM destina-se à teste de programas escritos na linguagem C, entretanto, é uma ferramenta multilinguagem que permite a configuração para outras linguagens.

A ferramenta possui 71 operadores de mutação divididos em quatro classes: mutação de comandos, mutação de operadores, mutação de variáveis e mutação de constantes. Essa divisão permite a escolha dos operadores de mutação de acordo com a classe de erros que se deseja revelar, de modo que a geração dos mutantes possa ser feita em etapas ou que se possa dividir a atividade de teste entre vários testadores (Vincenzi, 1998).

A ferramenta permite ao testador avaliar a adequação de um conjunto de casos de teste **T** para um determinado programa **P** e com o resultado dessa avaliação o testador pode melhorar o conjunto **T** visando a satisfazer o critério Análise de Mutantes.

As operações mínimas suportadas pela ferramenta são:

- Manipulação de casos de teste: execução, inclusão e exclusão.
- Manipulação de mutantes: geração, execução e análise.
- Análise de adequação: escore de mutação e relatórios estatísticos.

A ferramenta *PROTEUM* pode importar casos de teste da ferramenta *POKETOOL*, de sessões de teste dela mesma ou pode gerar casos de teste aleatoriamente. As informações geradas pela *PROTEUM* são armazenadas em uma base de dados composta por: uma base com informações sobre os casos de teste utilizados, uma base com informações sobre os mutantes e alguns arquivos intermediários que descrevem o programa em teste. Uma sessão de teste caracteriza-se de uma seqüência de operações realizadas sobre essas bases de dados.

A implementação da ferramenta *PROTEUM* facilitou a condução de trabalhos comparativos entre critérios de teste funcionais, estruturais e baseados em erros, assim como o estudo e a avaliação de estratégias de aplicação de variantes da Análise de Mutantes (Mutação Restrita) a exemplo de Mathur e Wong (1993), no contexto de programas C, linguagem de muito interesse no ambiente UNIX. Assim, viabiliza a condução de *benchmarks* que visem a demonstrar a factibilidade desse critério, avaliar o custo e a sua eficiência e avaliar o aspecto complementar deste com outros critérios.

A ferramenta *PROTEUM/IM*

A *PROTEUM/IM* apóia o critério *Interface Mutation*. A ferramenta está disponível para os sistemas operacionais SunOS e Linux e configurada para o teste de programas escritos na linguagem C, sendo também uma ferramenta multilinguagem (Delamaro, 1997a). A *PROTEUM/IM* possui 33 operadores de mutação divididos em dois grupos. O primeiro grupo aplica mutações em um módulo B chamado por um outro módulo A, enquanto o segundo grupo aplica mutações no ponto onde B é chamado dentro de A.

A ferramenta permite ao testador avaliar a adequação de um conjunto de casos de teste **T** para um determinado programa **P** e com o resultado dessa avaliação o testador pode melhorar o conjunto **T** visando a satisfazer o critério Mutação de Interface.

As operações mínimas suportadas pela ferramenta são as mesmas suportadas pela PROTEUM:

- Manipulação de casos de teste: execução, inclusão e exclusão.
- Manipulação de mutantes: geração, execução e análise.
- Análise de adequação: escore de mutação e relatórios estatísticos.

O Quadro 2.1 mostra quais são os programas que compõem as ferramentas *PROTEUM/IM* e *PROTEUM* (Vincenzi, 1998). Esses programas podem ser utilizados em conjunto na condução de uma sessão de teste. Os programas disponíveis estão divididos em dois grupos. O primeiro é composto por programas básicos que agem diretamente na base de teste que caracteriza a sessão. O segundo é composto por programas utilitários que utilizam os programas básicos para realizar algumas operações durante uma sessão de teste (Delamaro e Maldonado, 1997c).

A execução desses programas pode ser feita de duas formas: diretamente na linha de comando através de *scripts* ou de forma transparente para o usuário através da interface gráfica. A interface gráfica facilita a condução de uma sessão de teste no caso do usuário ser iniciante. Os recursos de visualização dos casos de teste e dos mutantes são melhores com a interface gráfica, tornando a condução da sessão de teste mais fácil. No entanto, esse tipo de interface depende muito da interação do usuário, sendo, assim, menos flexível do que chamar diretamente os programas (Vincenzi, 1998). A chamada direta aos programas, na forma de *scripts*, reduz o número de interações com as ferramentas e possibilita a execução de longas sessões de teste em *batch*, nas quais o usuário pode construir um programa especificando as tarefas a serem

realizadas e a ferramenta executa esse programa, reduzindo, assim, o tempo gasto na atividade de teste. Entretanto, se o usuário optar por utilizar *scripts*, a elaboração de *scripts* exige um esforço de programação e um completo domínio tanto dos conceitos sobre teste baseado em mutação quanto dos próprios programas que compõem a ferramenta.

Quadro 2.1 - Programas que compõem a PROTEUM/IM e a PROTEUM.

Programas Básicos	
li	Transforma um programa C para uma linguagem intermediária (LI)
li2nli	Cria o grafo de programa e adiciona informações sobre os nós na LI
ptest	Cria e manipula arquivos de teste de programas, os quais descrevem as características gerais da sessão de teste
tcase	Cria e manipula a base de dados de casos de teste
muta	Cria e manipula a base de dados dos mutantes
exemuta	Constrói o código fonte dos mutantes e os executa; ativa/desativa mutantes
opmuta	Aplica os operadores de mutação no programa original, criando descritores de mutação
report	Cria um relatório sobre a efetividade dos casos de teste
Programas Utilitários	
test-view	Cria uma nova sessão de teste
tcase-add	Insere um caso de teste interativamente
muta-gen	Gera descritores de mutação e os insere na base de dados dos mutantes
muta-view	Permite visualizar e analisar mutantes

A ferramenta *POKETOOL*

A *POKETOOL* (***Po****tential Uses* ***Cr****iteria* ***Tool*** *for Program Testing*) (Chaim, 1991) é uma ferramenta de teste que apóia os critérios estruturais Potenciais-Usos, Todos-Nós e Todos-Arcos. A versão inicial da ferramenta utiliza unidades de programas escritos na linguagem C, entretanto, a ferramenta possui a característica de multilinguagem.

A ferramenta é orientada à sessão de trabalho, na qual o usuário entra com o programa a ser testado, com o conjunto de dados de teste e seleciona um dos critérios suportados.

A *POKETOOL* possui nove funções para fornecer os dados acima e são mostradas no Quadro 2.2.

A ferramenta *ATAC*

A *ATAC* (*Automatic Testing Analysis tool in C*) realiza testes baseando-se na cobertura do fluxo de dados de programas escritos na linguagem C. Dado um programa para teste, a *ATAC* encontra o seu conjunto de atributos testáveis tais como blocos, comandos de decisão, p-uso, c-uso e todos-usos, e instrumenta o programa para armazenar informações através de *trace* durante a execução do teste. A *ATAC* utiliza esse *trace* para gerar os casos de teste (Wong *et al.*, 1997a).

A ferramenta conduz a seleção dos casos de teste baseando-se nas modificações realizadas e encontra os conjuntos minimizados e priorizados de casos de teste. Para o conjunto priorizado, a ferramenta seleciona casos de teste que melhoram a cobertura com respeito a um determinado critério e para o conjunto minimizado a ferramenta utiliza um algoritmo de minimização.

Quadro 2.2 – Funções implementadas pela ferramenta POKETOOL.

Funções da *POKETOOL*	
Grafo de Fluxo de Controle	Gera o grafo de fluxo de controle (GFC) do programa em teste. O GFC é gerado a partir de uma versão em linguagem intermediária (LI) produzida pela ferramenta a partir do programa em teste
Cálculo dos Arcos Primitivos	Calcula os arcos primitivos do GFC que irão servir para geração da versão instrumentada do programa em teste
Extensão do Grafo de Fluxo de Controle	Gera o Grafo Def
Interface Gráfica	Apresenta o GFC (Vilela, 1994; Vilela *et al*., 1996)
Instrumentação	Insere comandos de escrita (pontas de prova) no programa em teste para cada nó do GFC que produzem um *trace* da execução dos casos de teste.
Compilador Selecionado	Compilador da linguagem fonte na qual o programa em teste foi implementado
Execução do Programa	Controla a execução do programa em teste
Construção do Grafo(i)	Gera o conjunto de caminhos e associações requeridas para satisfazer os critérios Potenciais-Usos
Avaliação	Verifica se o conjunto de caminhos ou associações executados satisfaz o critério selecionado, fazendo um relacionamento de cobertura

As ferramentas *DEJAVU* e *RePOKETOOL*

Considerando o teste de regressão, algumas técnicas são apoiadas por ferramentas. Rothermel e Harrold (1997 e 1998) propuseram uma técnica de teste de regressão que se baseia na cobertura para selecionar casos de teste e é implementada pela uma ferramenta *DEJAVU*. Granja (1997) desenvolveu uma técnica seletiva que se baseia no teste estrutural mais especificamente nos critérios Potenciais-Usos que apoiada pela ferramenta *RePOKETOOL*. A *RePOKETOOL* (**Re**gression Testing support for **Po**tencial-Uses **Cri**teria **Tool**) pode ser aplicada em unidades que serão modificadas. A ferramenta seleciona casos de teste para serem utilizados no teste de regressão a partir dos elementos requeridos para o teste estrutural que foram modificados ou inseridos após a fase de manutenção.

2.6.2 Estudos Empíricos

Os critérios de adequação fornecem mecanismos para seleção sistemática de um subconjunto de casos de teste para um programa, de modo que esse subconjunto seja eficaz na detecção de defeitos. A partir de um dado critério de adequação pode-se obter um número infinito de subconjuntos de casos de teste adequados ao programa em teste. Dependendo do critério e da forma utilizada na geração dos casos de teste, pode-se chegar a um conjunto bastante grande, difícil de ser manipulado e que aumenta o custo na condução da atividade de teste (Vincenzi, 1998).

"A minimização de conjuntos de casos de teste tem o objetivo de a partir de um conjunto de casos de teste **T** gerar um conjunto $\mathbf{T_{min}}$, a partir de um conjunto de casos de teste T, com o mesmo grau de adequação de **T**, geralmente com um tamanho menor. A redução do tamanho do conjunto **T** é obtida pela eliminação de casos de teste redundantes e obsoletos. A minimização procura beneficiar os testes de regressão, como também fornecer um mecanismo mais real para avaliação do custo de aplicação de um critério. Muitas vezes esse tipo de avaliação é que determina o uso ou não de um critério para a condução dos testes" (Souza, 1996).

As estratégias que realizam minimização de conjuntos de casos de teste, além de permitirem redução de custos nas atividades de teste, mais particularmente o critério Análise de Mutantes, também possibilitam a obtenção de parâmetros mais reais para quantificar o custo de aplicar um critério durante o desenvolvimento de estudos empíricos. Entretanto, a redução do conjunto é uma tarefa difícil na prática, uma vez que, segundo Garey e Johnson (1979), o problema de encontrar em conjunto mínimo é *NP-Completo.*

Uma contribuição relevante do trabalho de Souza (1996), foi a implementação de um módulo de minimização de casos de teste, denominado *MINIMIZE*, para a ferramenta *PROTEUM1.2-C*. Com base nesse módulo, um experimento foi realizado visando avaliar a redução no tamanho dos conjuntos e também o efeito dessa redução na eficácia em revelar erros. Observou-se que a minimização proporciona uma redução significativa no tamanho dos conjuntos de casos de teste e com relação à eficácia dos conjuntos mínimos houve, em alguns casos, uma pequena redução da eficácia. O módulo *MINIMIZE* foi modificado para se integrar a nova versão da ferramenta, *PROTEUM*1.4.1. No entanto, sua funcionalidade não foi alterada, mas sim a estrutura de dados. Essa adaptação do módulo foi realizada por Tullio (1999) em seu trabalho de iniciação científica.

O módulo *MINIMIZE* tem como principal objetivo obter um subconjunto de casos de teste a partir do conjunto inicial de casos de teste adequados ao critério Análise de Mutantes. A implementação do módulo está baseada no algoritmo proposto por Harrold *et al.* (1993) o qual apresenta a vantagem de poder ser aplicado a qualquer critério de teste que contenha associações entre os requisitos de teste e os casos de teste. A minimização é realizada sobre uma sessão de teste já existente, composta de um conjunto de casos de teste adequado ao critério Análise de Mutantes.

As principais características do módulo *MINIMIZE* são:

- Sendo um módulo integrante da ferramenta *PROTEUM,* este deve seguir o mesmo padrão de desenvolvimento e utilizar, quando possível, as bibliotecas já definidas. Além disso, desempenha função específica e funciona de forma isolada dos demais módulos, se relacionando somente através das bases de dados.
- Permite que seja mantido no conjunto minimizado um determinado número de casos de teste. Essa característica permite, por exemplo, manter os casos de teste que revelem erros.
- A chamada ao módulo pode ser feita via *shell scripts.*

Uma contribuição importante para a minimização de custos da aplicação dos critérios Análise de Mutantes e Mutação de Interface foram os trabalhos realizados por Barbosa (1998) e Vincenzi (1998). Barbosa desenvolveu um algoritmo que permite definir um conjunto de operadores essenciais para a Linguagem C, denominado ***Essencial***. Esse algoritmo estabelece um conjunto essencial de operadores considerando apenas o critério Análise de Mutantes. Para validar o algoritmo definido, utilizaram-se dois grupos distintos de programas: um conjunto de programas utilitários do UNIX e um conjunto de programas em C (27) definidos por Maldonado (1991). Para ambos os conjuntos de programas, o conjunto essencial obtido apresentou resultados bastante significativos quanto à redução de custos, com um decréscimo muito pequeno no grau de adequação em relação ao critério Análise de Mutantes (Barbosa, 1998).

Vincenzi desenvolveu uma estratégia de aplicação dos operadores de mutação, tanto para o critério Análise de Mutantes quanto para o critério Mutação de Interface. Essa estratégia estabelece o teste incremental que combina ambos os critérios de mutação. Com o desenvolvimento dessa estratégia, além de se objetivar a minimização dos custos da aplicação de ambos os critérios de mutação, pretendeu-se comparar a adequação entre os critérios Análise de

Mutantes e Mutação de Interface. A estratégia estabelecida foi validada por meio de experimentos realizados com os mesmos dois grupos de programas utilizados por Barbosa em seus experimentos.

2.7 Considerações Finais

Neste capítulo foram apresentadas as principais técnicas de teste de software e as principais características de cada técnica, além de diversas ferramentas que apoiam a aplicação dessas técnicas.

Tão importante quanto o teste realizado durante o processo de desenvolvimento de software é o teste de regressão, pois o software evolui constantemente e deve ser testado visando a garantir que suas funcionalidades não foram afetadas adversamente com as modificações. Pode-se perceber que cada técnica baseia-se em conjuntos diferentes de informações para derivar os requisitos de teste. Devido a esse fato, estudos empíricos são necessários visando a comparar a eficácia da aplicação de cada uma dessas técnicas.

Assim como o teste de software tradicional, as atividades de teste de regressão também demandam muito custo. Visando a reduzir esses custos, vários pesquisadores têm proposto diversas técnicas que selecionam um subconjunto de casos de teste, a partir do conjunto original, para realizar os testes de regressão. Além disso, pesquisadores têm estudado continuamente mecanismos para avaliar e comparar essas diversas técnicas.

O próximo capítulo apresenta os experimentos realizados com as Técnicas de teste de regressão baseadas em **Modificação** e **Mutação Seletiva.**

Capítulo 3

Aplicação e Avaliação das Técnicas de Teste de Regressão

3.1 Considerações Iniciais

A realização de estudos teóricos e empíricos é importante para fornecer conhecimento e subsídios para o estabelecimento de estratégias de teste de software, seja essa atividade realizada durante a fase de desenvolvimento ou durante a fase de manutenção. Assim como o teste de software realizado durante o desenvolvimento é apoiado por técnicas e critérios de teste, o teste de regressão realizado durante a fase de manutenção também pode ser apoiado por técnicas de teste que visam a selecionar um conjunto de casos de teste a serem reexecutados para avaliar a confiabilidade das modificações realizadas. Por meio de estudos empíricos, é possível avaliar o custo e os benefícios da aplicação das diversas técnicas de teste de regressão existentes na literatura. Essa avaliação visa a facilitar a escolha de uma dessas técnicas quando da realização das atividades de teste de regressão.

Assim, neste capítulo são descritos os estudos empíricos realizados com os programas utilitários do UNIX e o programa SPACE utilizando-se as técnicas de teste de regressão baseadas em Modificação e Mutação Seletiva. Neste trabalho a técnica baseada em Modificação foi aplicada em programas utilitários do UNIX, enquanto que a técnica baseada em Mutação Seletiva foi aplicada ao programa SPACE, ao contrário do trabalho realizado por Wong *et al.* como ilustrado na Figura 1.1 do Capítulo 1. O *framework* definido por Rothermel e Harrold (1996) foi utilizado para avaliar essas técnicas de teste de regressão.

3.2 Descrição Geral dos Experimentos

Rothermel e Harrold (1997) definem duas maneiras possíveis de se realizarem estudos empíricos relacionados ao teste de regressão: 1. Considerar que as versões modificadas são tentativas mal sucedidas de se evoluir o programa original; ou 2. Considerar o programa original como sendo a versão correta de uma família de versões incorretas (modificadas). Para estes experimentos, a primeira maneira foi utilizada.

O ambiente dos experimentos é constituído dos seguintes elementos:

- Programas em teste.
- Mecanismos e ferramentas.
- Métricas.

Os programas utilizados nos experimentos foram os programas utilitários do UNIX e o programa SPACE descritos no Capítulo 2. Os mecanismos e ferramentas utilizados foram: a ferramenta *PROTEUM*, a ferramenta *PROTEUM/IM* e o módulo de minimização *MINIMIZE*, também apresentados no Capítulo 2. As métricas utilizadas para a medição dos resultados, além das características do *framework* de Rothermel e Harrold descritas no Capítulo 2, foram as seguintes (Wong *et al.*, 1997a; Wong *et al.*, 1997b):

- **Redução de Tamanho**: [1 - (nº de casos selecionados em Ts/nº de casos em T)] *100
- **Eficácia:** (nº de defeitos revelados por Ts/nº de defeitos) * 100
- **Precisão**: (nº de casos de teste em Ts que revelam defeitos/nº casos de teste em Ts) * 100
- ***Recall***: (nº de casos de teste em Ts que revelam defeitos/nº casos de teste em T') * 100

Sendo que:

T = conjunto de casos de teste original
T_S = conjunto de casos teste de regressão selecionado.
T' = conjunto de casos de teste que revelam um comportamento diferente entre P e P'

As seguintes etapas foram realizadas e são descritas a seguir:

- **Etapa 1** - Aplicação e Avaliação da Técnica baseada em Modificação
- **Etapa 2** - Aplicação e Avaliação da Técnica baseada em Mutação Seletiva
- **Etapa 3** - Análise Comparativa dos Resultados obtidos com a Aplicação de ambas as Técnicas de Teste de Regressão com os resultados obtidos por Wong *et al.*(1997a, 1997b).
- **Etapa 4** - Avaliação das Técnicas de Teste de Regressão segundo o *Framework* de Rothermel e Harrold.

3.2.1 Aplicação e Avaliação da Técnica baseada em Modificação

Nesta Seção são descritos os passos realizados durante a aplicação da técnica baseada em Modificação nos programas utilitários do UNIX. Os seguintes passos foram realizados e são descritos a seguir:

- **Passo 1** - Preparação dos Dados dos Programas.
- **Passo 2** - Aplicação da Técnica baseada em Modificação nos programas utilitários do UNIX.
- **Passo 3** - Análise dos Resultados.

Passo 1 - Preparação dos Dados dos Programas

Os conjuntos de casos de teste utilizados para os programas utilitários do UNIX foram adquiridos do trabalho de Vincenzi (1998), pois para esses programas Vincenzi já havia definido um conjunto de casos de teste adequados em relação ao critério Análise de Mutantes para cada um dos cinco programas. Para cada um dos programas, um conjunto de programas modificados foi utilizado. Esses programas contêm apenas uma modificação e foram obtidos do trabalho de Wong *et al*, (1997b). Essas modificações podem ou não caracterizar um defeito, no entanto, o termo defeito será utilizado para referenciar uma modificação. A Tabela 3.1 apresenta o número de casos de teste e o número de modificações para cada programa. O Apêndice B apresenta os programas originais e suas modificações.

Utilizando-se um *script* escrito no *K-shell* do UNIX (Vergílio, 1997), uma **Tabela de Eficácia** contendo todos os defeitos que são revelados por cada um dos conjuntos de casos de teste foi gerada. Por meio dessa **Tabela de Eficácia** foi possível estabelecer níveis de dificuldade de detecção de cada defeito. Foram definidos quatro níveis. Defeitos que estão no nível I são revelados pelo intervalo de [0-25%) dos casos de teste; defeitos que estão no nível II são revelados por [25-50%) dos casos de teste; defeitos que estão no nível III são revelados por [50-75%) dos casos de teste; e defeitos que estão no nível IV são revelados por [75-100%] dos casos de teste (Wong *et al.*, 1997b). Defeitos que estão no nível I são os mais difíceis de serem revelados, enquanto que aqueles que estão no nível IV são os mais fáceis de serem revelados. A Tabela 3.2 mostra em que nível está cada um dos defeitos de programa, em relação aos conjuntos de casos de teste disponíveis.

Tabela 3.1 - Dados dos programas utilitários do UNIX.

Programas	Conjunto de Casos de Teste (T)	Número de Defeitos (D)
Cal	184	20
Checkeq	213	22
Comm	801	19
Look	253	20
Uniq	510	19

Tabela 3.2 – Categorização dos defeitos dos programas utilitários do UNIX.

Programas	Nível I	Nível II	Nível III	Nível IV
Cal	D2 D5 D6 D9 D18 D20	D4 D11 D12 D19	D3 D7 D8 D10 D13 D14 D15 D16 D17	D1
Checkeq	D1 D6 D11 D13 D20	D4 D19 D22	D2 D3 D8 D9 D12 D14 D15 D16 D18 D21	D5 D7 D10 D17
Comm	D2 D3 D4 D6 D7 D10 D12 D13 D14 D16 D17 D19	D5 D8 D15 D18	D9 D11	D1
Look	D1 D3 D4 D5 D6 D7 D8 D9 D10 D11 D12 D13 D14 D15 D18 D19 D20	D16	D2 D17	-
Uniq	D2 D3 D5 D6 D9 D10 D11 D14 D15 D16 D18 D19	D7 D8 D12 D13 D17	D1 D4	-

O símbolo "-" significa que não existem defeitos nesse nível.

Para cada um dos programas, a precisão e a eficácia do conjunto de casos de teste T foi calculada. A Tabela 3.3 mostra esses dados. Para o programa Checkeq, o conjunto T não é capaz de revelar todos os defeitos disponíveis. Para os programas Comm, Look e Uniq, os defeitos não revelados pelo conjunto T na verdade não são defeitos, são programas equivalentes ao programa

Assim, qualquer que seja o conjunto de casos de teste, esse não será capaz de causar um comportamento diferente entre os programas originais e os modificados. Nesse caso, para essas modificações que não caracterizam defeitos, não foram selecionados conjuntos de casos de teste de regressão e a eficácia dos conjuntos de casos de teste para esses programas está relacionada às modificações que efetivamente se caracterizam como defeitos. Para o programa Checkeq, um novo caso de teste foi gerado durante o teste de regressão para relevar o defeito que faltava. Portanto, a eficácia do conjunto de casos de teste T para esse programa não é de 100%. Com relação à precisão, analisando-se a Tabela 3.3, pode-se notar que somente no caso do programa Checkeq todos os casos de teste disponíveis revelam pelo menos um defeito, pois a precisão foi de 100%. O programa Uniq é aquele que possui, proporcionalmente, o menor número de casos de teste que revelam defeitos, ou seja, 21,858% dos casos de teste não revelam defeitos.

Tabela 3.3 – Precisão e eficácia dos conjuntos de casos de teste T dos programas utilitários do UNIX.

Programas	Precisão	Eficácia
Cal	98,370	100
Checkeq	100	95,455
Comm	99,625	100
Look	88,142	100
Uniq	78,431	100

Passo 2 - Aplicação da Técnica baseada em Modificação nos Programas utilitários do UNIX

As seguintes tarefas foram executadas e são descritas a seguir:

- **Tarefa 1** – Definição do Conjunto T' (*modification-revealing*) com relação a cada defeitos.
- **Tarefa 2** – Definição do Conjunto T" (*modification-traversing*) com relação aos programas originais e seus defeitos.
- **Tarefa 3** – Seleção dos Conjuntos de Casos de Teste de Regressão para cada uma das Modificações.

Tarefa 1 - Definição do Conjunto T', modification-revealing, com relação a cada defeito

O conjunto T' foi definido, vide Tabela 3.4, utilizando-se a **Tabela de Eficácia** gerada no passo anterior para cada um dos programas. Esse conjunto é composto por todos os casos de teste *modification-revealing*.

Tabela 3.4 - Conjunto T' para cada defeito.

Defeito/Programa	Cal	Checkeq	Comm	Look	Uniq
D1	138	33	794	1	260
D2	1	151	3	183	82
D3	94	149	0	2	99
D4	89	101	26	0	338
D5	5	202	347	23	76
D6	38	0	24	31	85
D7	127	202	12	34	131
D8	127	138	233	39	236
D9	41	137	462	0	65
D10	107	159	3	0	72
D11	62	45	505	35	0
D12	90	153	125	3	142
D13	127	11	124	36	145
D14	127	140	198	37	60
D15	116	111	268	0	60
D16	127	141	79	114	78
D17	123	174	3	147	146
D18	29	147	248	2	21
D19	64	85	11	36	46
D20	32	37	--	6	--
D21	--	138	--	--	--
D22	--	83	--	--	--

Tarefa 2 - Definição dos Conjuntos T", modification-traversing, com relação aos programas originais e suas modificações

Os conjuntos T" são compostos pelos casos de teste *modification-traversing*. Esses conjuntos, evidentemente, contêm o conjunto T'. Para a definição desses conjuntos, foi necessário estabelecer o *trace* de execução de cada um dos casos de teste utilizando-se a ferramenta *PROTEUM/IM* (Delamaro, 1997a).

O código que sofre modificação em P para gerar P' foi identificado através do grafo de fluxo de controle dos programas. Em seguida, uma sessão de teste foi gerada, para cada um dos programas, e todos os casos de teste disponíveis executados com os programas originais. Um arquivo de trace para cada caso foi gerado, possibilitando encontrar o caminho percorrido pelo caso de teste em P e verificar se esse caso de teste passava por alguma modificação. Se o caso de teste exercitasse a modificação, esse caso de teste era incluído no conjunto T" referente à

modificação. Esses conjuntos, portanto, possuem todos os casos de teste *modification-traversing* com respeito a uma determinada modificação. Considerando o conjunto original T de cada programa, todo conjunto possui pelo menos um caso de teste que exercita uma modificação. Assim, a Tabela 3.5 mostra o número de casos de teste que estão nos conjuntos T" para cada um dos defeitos para cada programa.

Tabela 3.5 - Conjunto T" para cada defeito.

Defeito/Programa	Cal	Checkeq	Comm	Look	Uniq
D1	184	213	801	124	414
D2	1	162	800	183	285
D3	109	162	406	196	127
D4	89	110	406	173	363
D5	46	202	347	28	76
D6	38	202	50	196	369
D7	127	199	52	119	367
D8	127	147	452	133	637
D9	184	138	627	173	93
D10	127	199	627	143	367
D11	127	199	627	119	0
D12	100	153	252	196	251
D13	127	199	233	146	367
D14	127	140	534	196	93
D15	127	140	530	1	93
D16	127	141	746	196	261
D17	127	199	746	196	367
D18	38	199	746	116	45
D19	81	147	24	146	46
D20	38	37	--	40	--
D21	--	138	--	--	--
D22	--	170	--	--	--

Tarefa 3 – Seleção dos Conjuntos de Casos de Teste de Regressão para cada uma das Modificações

Após a definição de cada um dos conjuntos T", os mecanismos de minimização e priorização foram aplicados, conforme descrito a seguir.

Selecionando os conjuntos de casos de teste de regressão utilizando o mecanismo de minimização

Uma sessão de teste foi criada para cada um dos conjuntos de casos de teste *modification-traversing*, T", selecionados para cada um dos defeitos. Para cada um dos programas, os mutantes foram gerados e executados com os conjuntos de casos de teste T". A Tabela 3.6 sintetiza o número de mutantes gerados para cada um dos operadores de mutação essenciais. O

conjunto de operadores essenciais de mutação foi determinado por Barbosa (1998) em seu trabalho de mestrado a partir dos operadores de mutação definidos para a ferramenta PROTEUM. Em seguida, o módulo de minimização *MINIMIZE* foi aplicado a cada sessão de teste, gerando um conjunto de casos de teste de regressão minimizado, $T"_M$, para cada um dos defeitos de cada programa, preservando o mesmo índice de adequação em relação ao conjunto de operadores de mutação essenciais. A Tabela 3.7 apresenta o número de casos de teste em cada conjunto $T"_{MDx}$, sendo $T"_{MDx}$ o conjunto minimizado para o defeito Dx.

Tabela 3.6 – Número de mutantes gerados por operador de mutação essencial.

Operadores/Programas	Cal	Checkeq	Comm	Look	Uniq
SWDD	2	2	4	3	7
SMTC	8	4	5	7	9
SSDL	101	70	103	82	90
OLBN	12	33	9	12	15
ORRN	110	135	115	60	95
VTWD	124	130	50	74	44
VDTR	186	195	75	111	66
Cccr	939	340	164	125	108
Ccsr	645	540	51	115	42
Total Gerado	**2127**	**1449**	**576**	**589**	**476**
Total de Equivalentes	**204**	**126**	**102**	**118**	**43**

Tabela 3.7 – Número de casos de teste nos conjuntos minimizados $T"_{MDx}$.

Defeito/Programa	Cal	Checkeq	Comm	Look	Uniq
$T"_{MD1}$	19	23	22	18	10
$T"_{MD2}$	1	9	26	14	4
$T"_{MD3}$	12	9	0	16	1
$T"_{MD4}$	11	4	15	0	1
$T"_{MD5}$	10	22	12	8	1
$T"_{MD6}$	7	0	4	16	2
$T"_{MD7}$	11	22	4	13	6
$T"_{MD8}$	11	5	8	12	6
$T"_{MD9}$	19	4	8	0	2
$T"_{MD10}$	11	22	16	0	1
$T"_{MD11}$	11	23	16	12	0
$T"_{MD12}$	10	6	6	16	4
$T"_{MD13}$	11	22	6	13	2
$T"_{MD14}$	11	5	11	16	2
$T"_{MD15}$	11	7	12	0	6
$T"_{MD16}$	11	6	22	16	2
$T"_{MD17}$	11	22	22	16	2
$T"_{MD18}$	8	22	6	18	1
$T"_{MD19}$	8	5	6	13	1
$T"_{MD20}$	7	5	--	8	--
$T"_{MD21}$	--	4	--	--	--
$T"_{MD22}$	--	16	--	--	--

Selecionando os conjuntos de casos de teste de regressão utilizando um mecanismo de priorização baseado na eficácia do caso de teste

Com o auxílio da Tabela de Eficácia, os conjuntos de casos de teste *modification-traversing* de cada um dos defeitos dos programas foram ordenados de acordo com a eficácia dos casos de teste: casos de teste que mais revelavam defeitos foram colocados no topo da lista de prioridades. Em seguida, esses casos de teste com maior prioridade foram sendo selecionados para os conjuntos de casos de teste priorizados $T"_{PDx}$ até que o defeito fosse revelado, sendo $T"_{PDx}$ o conjunto priorizado para o defeito Dx. Esse processo foi realizado para todos os defeitos de cada um dos programas utilitários do UNIX. Em estudos empíricos realizados por Rothermel *et al.* (1999) comparando diferentes mecanismos de priorização, o mecanismo utilizado neste trabalho mostrou-se mais eficiente que outros mecanismos avaliados. A Tabela 3.8 mostra o número de casos de teste selecionados pelo mecanismo de priorização para cada defeito de cada programa.

Em geral, exceto para os programas Cal e Uniq, o tamanho dos conjuntos priorizados é menor que o tamanho dos conjuntos minimizados. Esse fato evidencia a importância de se ter informações com relação aos casos de teste disponíveis de forma a minimizar a relação custo/benefício.

Wong *et al.* utilizaram para a priorização informações sobre o incremento proporcionado na cobertura relativa a um determinado critério de teste. Essas informações podem ser consideradas para a definição de políticas de priorização. A decisão tomada neste trabalho de utilizar as informações sobre a eficácia dos casos de teste favorece a aplicação desta técnica do ponto de vista de custo/benefício, porém nem sempre essas informações estão disponíveis.

Tabela 3.8 – Número de casos de teste nos conjuntos priorizados $T"_{PDx}$.

Defeito/Programa	Cal	Checkeq	Comm	Look	Uniq
$T"_{MD1}$	12	12	4	1	5
$T"_{MD2}$	1	11	4	1	5
$T"_{MD3}$	5	11	0	1	2
$T"_{MD4}$	5	11	1	0	5
$T"_{MD5}$	8	11	4	1	1
$T"_{MD6}$	7	0	20	1	5
$T"_{MD7}$	12	11	4	1	5
$T"_{MD8}$	12	11	4	1	5
$T"_{MD9}$	12	11	4	0	5
$T"_{MD10}$	12	11	4	0	5
$T"_{MD11}$	12	11	1	1	0
$T"_{MD12}$	12	11	4	1	5
$T"_{MD13}$	12	11	4	1	5
$T"_{MD14}$	12	11	4	1	5
$T"_{MD15}$	12	11	4	0	5
$T"_{MD16}$	12	11	4	1	5
$T"_{MD17}$	12	11	4	1	5
$T"_{MD18}$	12	11	4	1	1
$T"_{MD19}$	12	11	4	1	1
$T"_{MD20}$	7	11	--	5	--
$T"_{MD21}$	--	11	--	--	--
$T"_{MD22}$	--	11	--	--	--

Passo 3 – Análise dos Resultados

A Tabela 3.9 mostra a média do número de casos de teste selecionados para os conjuntos minimizados e priorizados. As Tabelas 3.10 e 3.11 sintetizam os resultados dos experimentos apresentando a média dos valores de redução de tamanho, eficácia, precisão e *recall* correspondentes aos conjuntos minimizados e priorizados, respectivamente.

Observa-se que os valores de redução de tamanho foram significativos, variando entre 89,712 e 98,831% para os conjuntos minimizados, e entre 90,200 e 99,142% para os conjuntos priorizados. Analisando as tabelas, nota-se que 80% dos conjuntos priorizados obtiveram uma maior redução de tamanho com relação aos conjuntos minimizados. Apenas para o programa Uniq a redução dos conjuntos minimizados foi maior que a redução dos conjuntos priorizados. Para os programas Cal, Checkeq e Uniq, a diferença de redução de tamanho para os conjuntos minimizados e priorizados foi pequena, enquanto que para os programas Comm e Look essa diferença foi maior.

A precisão dos conjuntos priorizados foi melhor que a precisão dos conjuntos minimizados, sendo a maior diferença para os valores do programa Checkeq, 47,657%, e a menor para os valores do programa Uniq, 10,698%. Já os valores de *recall* foram semelhantes

para todos os conjuntos minimizados e priorizados, apresentando apenas uma pequena variação entre os valores, sendo a maior de 5,808% para o programa Look. É importante ressaltar que a precisão e o *recall* são medidas relativas diretamente dependentes das características do conjunto de casos de teste disponível para realizar a atividade de teste de regressão. Quanto à eficácia, não houve redução dos valores tanto para os conjuntos minimizados quanto para os conjuntos priorizados.

Nota-se que o uso da minimização baseada no conjunto de operadores essenciais de mutação produz bons resultados com relação aos resultados produzidos com a priorização, mesmo sem utilizar informações sobre a eficácia dos casos de teste. É importante ressaltar que com o mecanismo de priorização utilizado procurou-se evitar que casos de teste *non_modification-revealing* fossem selecionados, enquanto que com o mecanismo de minimização utilizado esse controle foi mais difícil de ser realizado, pois o algoritmo que o módulo *MINIMIZE* implementa seleciona aqueles casos de teste que mais "matam" mutantes sem considerar sua eficácia em revelar defeitos.

Tabela 3.9 – Média do número de casos de teste para os conjuntos minimizados $T"_{MDx}$ e para os conjuntos priorizados $T"_{PDx}$.

	Cal	Checkeq	Comm	Look	Uniq
$T"_{MDx}$	10,55	12,524	12,333	14,063	3,000
$T"_{PDx}$	10,05	11,048	4,556	1,25	4,167

Tabela 3.10 – Média para os conjuntos minimizados $T"_{MDx}$.

Programas	Redução de Tamanho	Eficácia	Precisão	Recall
Cal	89,712	100	59,815	16,293
Checkeq	92,251	95,455	43,613	10,592
Comm	97,419	100	48,636	9,737
Look	90,343	100	73,315	27,243
Uniq	98,831	100	78,191	1,838

Tabela 3.11 – Média para os conjuntos priorizados $T"_{PDx}$.

Programas	Redução de Tamanho	Eficácia	Precisão	Recall
Cal	90,200	100	90,625	18,159
Checkeq	93,164	95,455	91,270	9,828
Comm	99,046	100	72,778	13,740
Look	99,142	100	93,750	21,435
Uniq	98,377	100	88,889	3,723

A seguir, os resultados com relação a cada programa utilitário do UNIX são apresentados.

As Tabelas 3.12 e 3.13 apresentam os resultados obtidos para cada um dos conjuntos minimizados e priorizados, respectivamente.

Analisando as tabelas, é possível notar que tanto os conjuntos minimizados quanto os conjuntos priorizados obtiveram uma grande redução de tamanho sem afetar a eficácia dos conjuntos em revelar defeitos. Para 75% dos defeitos, os conjuntos minimizados foram menores ou iguais aos conjuntos priorizados.

Tanto para os conjuntos minimizados quanto para os conjuntos priorizados, o *recall* foi baixo e a precisão foi alta para a maioria dos defeitos. No entanto, para o defeito D2 tanto o *recall* quanto a precisão foram altos, sendo de 100%. Isso ocorreu, pois no caso do defeito D2 somente um caso de teste do conjunto original é capaz de executá-lo e revelá-lo, e tanto o mecanismo de minimização quanto o de priorização foram capazes de selecionar esse caso de teste. Esse fato também explica a não redução no tamanho dos conjuntos minimizados e priorizados. No caso dos conjuntos priorizados, a maioria obteve 100% de precisão significando que todos os casos de teste selecionados pelo mecanismo de priorização são capazes de revelar o defeito. É possível notar que muitas vezes o valor do *recall* e da redução de tamanho são complementares, atingindo 100%. Esse fato ocorre quando a quantidade de casos de teste que revelam e exercitam um defeito é o mesmo, como, por exemplo, para os defeitos D13 e D14.

Tabela 3.12 – Cal: Resultados obtidos para cada um dos conjuntos minimizados $T"_{MDx}$.

Defeito	Redução de Tamanho	Eficácia	Precisão	*Recall*
$T"_{MD1}$	89,674	100	57,895	7,971
$T"_{MD2}$	0	100	100	100
$T"_{MD3}$	88,991	100	75,000	9,574
$T"_{MD4}$	87,640	100	100	12,360
$T"_{MD5}$	78,261	100	20,000	40
$T"_{MD6}$	81,579	100	100	18,421
$T"_{MD7}$	91,339	100	100	8,661
$T"_{MD8}$	91,339	100	100	8,661
$T"_{MD9}$	89,674	100	26,316	12,195
$T"_{MD10}$	91,339	100	72,727	7,477
$T"_{MD11}$	91,339	100	63,636	11,290
$T"_{MD12}$	90,000	100	70,000	7,778
$T"_{MD13}$	91,339	100	100	8,661
$T"_{MD14}$	91,339	100	100	8,661
$T"_{MD15}$	91,339	100	81,818	7,759
$T"_{MD16}$	91,339	100	100	8,661
$T"_{MD17}$	91,339	100	100	8,943
$T"_{MD18}$	78,947	100	50,000	13,793
$T"_{MD19}$	90,123	100	75,000	9,375
$T"_{MD20}$	81,579	100	71,429	15,625

Tabela 3.13 – Cal: Resultados obtidos para cada um dos conjuntos priorizados $T"_{PDx}$.

Defeito	Redução de Tamanho	Eficácia	Precisão	*Recall*
$T"_{PD1}$	93,478	100	41,667	3,623
$T"_{PD2}$	0	100	100	100
$T"_{PD3}$	95,413	100	100	5,319
$T"_{PD4}$	94,382	100	100	5,618
$T"_{PD5}$	82,609	100	12,500	20,000
$T"_{PD6}$	81,579	100	100	18,421
$T"_{PD7}$	90,551	100	100	9,449
$T"_{PD8}$	90,551	100	100	9,449
$T"_{PD9}$	93,478	100	75,000	21,951
$T"_{PD10}$	90,551	100	100	11,215
$T"_{PD11}$	90,551	100	100	19,355
$T"_{PD12}$	88,000	100	100	13,333
$T"_{PD13}$	90,551	100	100	9,449
$T"_{PD14}$	90,551	100	100	9,449
$T"_{PD15}$	90,551	100	100	10,345
$T"_{PD16}$	90,551	100	100	9,449
$T"_{PD17}$	90,551	100	100	9,756
$T"_{PD18}$	68,421	100	91,667	37,931
$T"_{PD19}$	85,185	100	91,667	17,188
$T"_{PD20}$	81,579	100	100	21,875

Programa CHECKEQ

Tanto para os conjuntos minimizados quanto para os conjuntos priorizados a redução de tamanho foi significativamente alta. Para 57,143% dos defeitos, os conjuntos minimizados foram menores que os conjuntos priorizados. Os valores de *recall* dos conjuntos minimizados foram, em geral, menores do que os valores de *recall* para os conjuntos priorizados. Somente para os defeitos D1, D5, D7, D13 e D17 os valores de *recall* dos conjuntos minimizados foram maiores do que os valores dos conjuntos priorizados. Para a maioria dos defeitos, exceto D1 e D12, a precisão dos conjuntos priorizados foi maior ou igual à precisão dos conjuntos minimizados. Tanto para os conjuntos minimizados quanto para os conjuntos priorizados não houve redução de eficácia.

Os resultados obtidos para os conjuntos minimizados e priorizados do programa Checkeq são mostrados nas Tabelas 3.14 e 3.15, respectivamente.

Tabela 3.14 – Checkeq: Resultados obtidos para cada um dos conjuntos minimizados T"$_{MDx}$.

Defeito	Redução de Tamanho	Eficácia	Precisão	*Recall*
T"$_{MD1}$	89,202	100	39,130	27,273
T"$_{MD2}$	94,444	100	88,889	5,298
T"$_{MD3}$	94,444	100	100	6,040
T"$_{MD4}$	96,364	100	75	2,970
T"$_{MD5}$	89,109	100	100	10,891
T"$_{MD6}$	-	-	-	-
T"$_{MD7}$	88,945	100	100	10,891
T"$_{MD8}$	96,599	100	100	3,623
T"$_{MD9}$	97,101	100	75	2,190
T"$_{MD10}$	88,945	100	45,455	6,289
T"$_{MD11}$	88,945	100	27,273	13,333
T"$_{MD12}$	96,078	100	100	3,922
T"$_{MD13}$	88,945	100	40,909	81,818
T"$_{MD14}$	96,429	100	100	3,571
T"$_{MD15}$	95,000	100	100	6,306
T"$_{MD16}$	95,745	100	100	4,255
T"$_{MD17}$	88,945	100	72,727	9,195
T"$_{MD18}$	88,945	100	22,727	3,401
T"$_{MD19}$	96,599	100	40	2,353
T"$_{MD20}$	86,486	100	100	13,514
T"$_{MD21}$	97,101	100	100	2,899
T"$_{MD22}$	90,588	100	12,500	2,410

Tabela 3.15 – Checkeq: Resultados obtidos para cada um dos conjuntos priorizados T"$_{PDx}$.

Defeito	Redução de Tamanho	Eficácia	Precisão	*Recall*
T"$_{PD1}$	94,366	100	8,333	3,030
T"$_{PD2}$	93,210	100	100	7,285
T"$_{PD3}$	93,210	100	100	7,383
T"$_{PD4}$	90	100	100	10,891
T"$_{PD5}$	94,554	100	100	5,446
T"$_{PD6}$	-	-	-	-
T"$_{PD7}$	94,472	100	100	5,446
T"$_{PD8}$	92,517	100	100	7,971
T"$_{PD9}$	92,029	100	100	8,029
T"$_{PD10}$	94,472	100	100	6,918
T"$_{PD11}$	94,472	100	100	24,444
T"$_{PD12}$	92,810	100	8,333	7,190
T"$_{PD13}$	94,366	100	100	9,091
T"$_{PD14}$	92,143	100	100	7,857
T"$_{PD15}$	92,143	100	100	9,910
T"$_{PD16}$	92,199	100	100	7,801
T"$_{PD17}$	94,472	100	100	6,322
T"$_{PD18}$	94,472	100	100	7,483
T"$_{PD19}$	92,517	100	100	12,941
T"$_{PD20}$	70,270	100	100	29,730
T"$_{PD21}$	92,029	100	100	7,971
T"$_{PD22}$	93,529	100	100	13,253

Programa COMM

Para o programa Comm, os conjuntos priorizados obtiveram uma maior redução de tamanho com relação aos conjuntos minimizados. Para 88,889% dos defeitos, os conjuntos priorizados foram menores que os conjuntos minimizados. Somente para os defeitos D6 e D7, os conjunto minimizados foram menores. Os valores de precisão dos conjuntos priorizados foram maiores para todos os defeitos com relação aos conjuntos minimizados. Tanto os conjuntos minimizados quanto os conjuntos priorizados obtiveram valores semelhantes de *recall*, sendo esses valores muitas vezes o mesmo para alguns defeitos, como é o caso dos defeitos D2, D4, D10, D12 e D17. Não houve redução de eficácia tanto para os conjuntos minimizados e quanto para os conjuntos priorizados.

Os resultados para os conjuntos minimizados e priorizados do programa Comm estão descritos nas Tabelas 3.16 e 3.17, respectivamente.

Tabela 3.16 – Comm: Resultados obtidos para cada um dos conjuntos minimizados $T"_{MDx}$.

Defeito	Redução de Tamanho	Eficácia	Precisão	*Recall*
$T"_{MD1}$	96,629	100	92,593	3,149
$T"_{MD2}$	96,750	100	3,846	33,333
$T"_{MD3}$	-	-	-	-
$T"_{MD4}$	96,305	100	6,667	3,846
$T"_{MD5}$	96,542	100	91,667	3,170
$T"_{MD6}$	92	100	50	8,333
$T"_{MD7}$	92,308	100	25	8,333
$T"_{MD8}$	98,230	100	75	2,575
$T"_{MD9}$	98,724	100	87,500	1,515
$T"_{MD10}$	97,448	100	6,250	33,333
$T"_{MD11}$	97,448	100	93,750	2,970
$T"_{MD12}$	97,619	100	16,667	0,800
$T"_{MD13}$	97,425	100	33,333	1,613
$T"_{MD14}$	97,940	100	54,545	3,030
$T"_{MD15}$	97,736	100	91,667	4,104
$T"_{MD16}$	97,051	100	9,091	2,532
$T"_{MD17}$	97,051	100	4,545	33,333
$T"_{MD18}$	98,739	100	83,333	2,016
$T"_{MD19}$	75	100	50	27,273

Tabela 3.17 – Comm: Resultados obtidos para cada um dos conjuntos priorizados $T"_{PDx}$.

Defeito	Redução de Tamanho	Eficácia	Precisão	*Recall*
$T"_{PD1}$	99,501	100	100	0,504
$T"_{PD2}$	99,375	100	20	33,333
$T"_{PD3}$	-	-	-	-
$T"_{PD4}$	99,507	100	50	3,846
$T"_{PD5}$	98,847	100	100	1,153
$T"_{PD6}$	60	100	90	75
$T"_{PD7}$	90,385	100	40	16,667
$T"_{PD8}$	99,115	100	100	1,717
$T"_{PD9}$	99,362	100	100	0,866
$T"_{PD10}$	99,203	100	20	33,333
$T"_{PD11}$	99,362	100	100	0,792
$T"_{PD12}$	99,206	100	50	0,800
$T"_{PD13}$	98,283	100	100	3,226
$T"_{PD14}$	99,251	100	100	2,020
$T"_{PD15}$	99,245	100	100	1,493
$T"_{PD16}$	99,330	100	20	1,266
$T"_{PD17}$	99,330	100	20	33,333
$T"_{PD18}$	99,160	100	100	1,613
$T"_{PD19}$	83,333	100	100	36,364

Programa LOOK

O programa Look foi o programa que obteve uma maior diferença entre os valores de redução de tamanho dos conjuntos minimizados e priorizados com relação aos outros programas utilitários do UNIX. Para todos os conjuntos priorizados, a redução de tamanho foi maior. Os valores de precisão para os conjuntos priorizados foram maiores que os valores de precisão dos conjuntos minimizados, atingindo 100% para a maioria dos defeitos. Os valores de *recall*, em geral, foram menores para os conjuntos priorizados. Não houve redução de eficácia tanto para os conjuntos minimizados quanto para os conjuntos priorizados.

As Tabelas 3.18 e 3.19 mostram os resultados para os conjuntos minimizados e priorizados, respectivamente.

Tabela 3.18 – Look: Resultados obtidos para cada um dos conjuntos minimizados $T"_{MDx}$.

Defeito	Redução de Tamanho	Eficácia	Precisão	*Recall*
$T"_{MD1}$	85,484	100	5,556	100
$T"_{MD2}$	92,350	100	100	7,650
$T"_{MD3}$	91,837	100	6,250	50
$T"_{MD4}$	-	-	-	-
$T"_{MD5}$	71,429	100	37,500	13,043
$T"_{MD6}$	91,837	100	31,250	16,129
$T"_{MD7}$	89,076	100	53,846	20,588
$T"_{MD8}$	90,977	100	58,333	17,949
$T"_{MD9}$	-	-	-	-
$T"_{MD10}$	-	-	-	-
$T"_{MD11}$	89,916	100	75	25,714
$T"_{MD12}$	91,837	100	6,250	33,333
$T"_{MD13}$	81,096	100	61,538	22,222
$T"_{MD14}$	91,837	100	50	21,622
$T"_{MD15}$	-	-	-	-
$T"_{MD16}$	91,837	100	75	10,526
$T"_{MD17}$	91,837	100	50	5,442
$T"_{MD18}$	84,483	100	5,556	50
$T"_{MD19}$	91,096	100	69,231	25
$T"_{MD20}$	80	100	12,500	16,667

Tabela 3.19 – Look: Resultados obtidos para cada um dos conjuntos priorizados $T"_{PDx}$.

Defeito	Redução de Tamanho	Eficácia	Precisão	*Recall*
$T"_{PD1}$	98,387	100	50	100
$T"_{PD2}$	99,454	100	100	0,546
$T"_{PD3}$	98,980	100	50	50
$T"_{PD4}$	-	-	-	-
$T"_{PD5}$	99,429	100	100	4,348
$T"_{PD6}$	99,490	100	100	3,226
$T"_{PD7}$	99,160	100	100	2,941
$T"_{PD8}$	99,248	100	100	2,564
$T"_{PD9}$	-	-	-	-
$T"_{PD10}$	-	-	-	-
$T"_{PD11}$	99,160	100	100	2,857
$T"_{PD12}$	99,490	100	100	33,333
$T"_{PD13}$	99,315	100	100	2,778
$T"_{PD14}$	99,490	100	100	2,703
$T"_{PD15}$	-	-	-	-
$T"_{PD16}$	99,490	100	100	0,877
$T"_{PD17}$	99,490	100	100	0,680
$T"_{PD18}$	99,138	100	100	50
$T"_{PD19}$	99,315	100	100	2,778
$T"_{PD20}$	87,500	100	100	83,333

Programa UNIQ

Tanto para os conjuntos minimizados quanto para os conjuntos priorizados, o programa Uniq obteve uma alta redução de tamanho dos conjuntos e baixos valores para o *recall*. O programa Uniq foi o único para o qual a maioria dos conjuntos minimizados obteve uma maior redução de tamanho com relação aos conjuntos priorizados. Para 77,778% dos defeitos, os conjuntos minimizados foram menores ou iguais aos conjuntos priorizados. A precisão dos conjuntos priorizados foi, em geral, maior que a precisão para os conjuntos minimizados, atingindo 100% para a maioria dos defeitos. Não houve redução de eficácia tanto para os conjuntos minimizados e quanto para os conjuntos priorizados.

As Tabelas 3.20 e 3.21 mostram os resultados obtidos com os conjuntos minimizados e priorizados, respectivamente.

Tabela 3.20 – Uniq: Resultados obtidos para cada um dos conjuntos minimizados $T"_{MDx}$.

Defeito	Redução de Tamanho	Eficácia	Precisão	*Recall*
$T"_{MD1}$	97,585	100	60	2,308
$T"_{MD2}$	98,596	100	100	1,220
$T"_{MD3}$	99,213	100	100	1,010
$T"_{MD4}$	99,725	100	100	0,296
$T"_{MD5}$	98,684	100	100	1,124
$T"_{MD6}$	99,458	100	50	1,176
$T"_{MD7}$	98,365	100	16,667	0,763
$T"_{MD8}$	98,365	100	33,333	0,847
$T"_{MD9}$	97,849	100	100	3,077
$T"_{MD10}$	99,728	100	100	1,389
$T"_{MD11}$	-	-	-	-
$T"_{MD12}$	98,406	100	25	0,704
$T"_{MD13}$	99,455	100	50	0,690
$T"_{MD14}$	97,849	100	100	3,333
$T"_{MD15}$	93,548	100	66,667	6,667
$T"_{MD16}$	99,234	100	50	1,282
$T"_{MD17}$	99,455	100	50	0,685
$T"_{MD18}$	97,778	100	100	4,762
$T"_{MD19}$	97,826	100	100	1,754

Tabela 3.21 – Uniq: Resultados obtidos para cada um dos conjuntos priorizados $T"_{PDx}$.

Defeito	Redução de Tamanho	Eficácia	Precisão	*Recall*
$T"_{PD1}$	98,792	100	100	1,923
$T"_{PD2}$	98,246	100	60	3,659
$T"_{PD3}$	98,425	100	100	2,020
$T"_{PD4}$	98,623	100	100	1,479
$T"_{PD5}$	98,684	100	100	1,124
$T"_{PD6}$	98,645	100	40	2,353
$T"_{PD7}$	98,638	100	80	3,053
$T"_{PD8}$	98,638	100	100	2,119
$T"_{PD9}$	94,624	100	100	7,692
$T"_{PD10}$	98,638	100	60	4,167
$T"_{PD11}$	-	-	-	-
$T"_{PD12}$	98,008	100	100	3,521
$T"_{PD13}$	98,638	100	100	3,448
$T"_{PD14}$	94,624	100	100	8,333
$T"_{PD15}$	94,624	100	100	8,333
$T"_{PD16}$	98,084	100	60	3,846
$T"_{PD17}$	98,638	100	100	3,425
$T"_{PD18}$	97,778	100	100	4,762
$T"_{PD19}$	97,826	100	100	1,754

3.2.2 Aplicação e Avaliação da Técnica baseada em Mutação Seletiva

Nesta Seção são descritos os passos realizados durante a aplicação da técnica baseada em Mutação Seletiva no programa SPACE. Relembrando, Wong *et al.* utilizaram-se dos programas utilitários do UNIX para aplicar a técnica baseada em Mutação Seletiva.

Os seguintes passos foram realizados e são descritos a seguir:

- **Passo 1** - Preparação dos dados do Programa.
- **Passo 2** - Definição dos Operadores de Mutação.
- **Passo 3** - Aplicação da Técnica baseada em Mutação Seletiva no programa SPACE.
- **Passo 4** - Análise dos Resultados obtidos.

Passo 1 – Preparação dos dados do Programa

Algumas modificações no código fonte do programa SPACE foram realizadas para que o mesmo se adequasse à ferramenta *PROTEUM1.4.1*. Essas modificações foram realizadas na passagem de parâmetro das funções devido a algumas limitações da ferramenta em lidar com o

tipo de passagem de parâmetro que estava sendo utilizada no programa. Foram modificadas todas as funções que compõem o programa. A Tabela 3.22 mostra os dados relacionados ao programa SPACE.

Tabela 3.22 - Dados do Programa SPACE.

Programa	Conjunto de Casos de Teste (T)	Número de Defeitos (D)
Space	1000	10

Assim como realizado com os programas utilitários do UNIX, para os defeitos do programa SPACE, por meio da **Tabela de Eficácia,** estabeleceram-se os níveis de categorização para os dez defeitos disponíveis para o programa. Os mesmos níveis de categorização definidos para os programas utilitários do UNIX foram definidos para o programa SPACE de acordo com o conjunto de casos de teste disponível: Nível I - 0-25% (dos casos de teste); Nível II - 25-50% (dos casos de teste); Nível III - 50-75% (dos casos de teste); e Nível IV - 75-100% (dos casos de teste). A Tabela 3.23 mostra em qual nível um determinado defeito está.

Para o programa SPACE as versões modificadas foram geradas a partir da ativação dos defeitos no programa original. Ativado o defeito, um programa executável da versão modificada foi gerado e utilizado para o experimento.

Tabela 3.23 – Categorização dos Defeitos do Programa SPACE.

Programa	Nível I	Nível II	Nível III	Nível IV
Space	D1 D2 D3 D4 D5 D6 D7 D8 D9	D10	-	-

O símbolo "-" significa que não existem defeitos nesse nível.

Como se pode notar, a maioria dos defeitos disponíveis são difíceis de serem revelados, 90% dos defeitos estão no nível I. Analisando a **Tabela de Eficácia** gerada, 494 casos de teste, dos 1000 disponíveis, são capazes de revelar defeitos. Assim, a precisão do conjunto de casos de teste original é de 49,4%. Entretanto, dos dez defeitos disponíveis, o defeito D8 não foi revelado pelo conjunto de casos de teste original. Como esse defeito não é revelado pelo conjunto T, a eficácia desse conjunto é de 90%. A Tabela 3.24 mostra o número de casos de teste que revelam um determinado defeito.

Tabela 3.24 – Número de Casos de Teste que revelam os Defeitos do Programa SPACE.

D1	D2	D3	D4	D5	D6	D7	D8	D9	D10
18	18	106	3	3	19	19	-	29	408

Passo 2 – Definição dos Operadores de Mutação

Os operadores de mutação utilizados nos experimentos são os operadores essenciais determinados por Barbosa (1998) em seu trabalho de mestrado. Barbosa determinou um conjunto com 9 operadores de mutação a partir dos 71 operadores da ferramenta *PROTEUM*. O conjunto de operadores de mutação essenciais inclui alguns dos operadores de mutação utilizados por Wong *et al.*(1997b). A Figura 3.1 mostra a relação entre ambos os conjuntos de operadores de mutação, e a Tabela 3.25 mostra os operadores de mutação essenciais que estão sendo utilizados e o número de mutantes gerados por cada um deles para o programa SPACE. A Tabela 3.26 mostra o número de mutantes que são gerados pelos operadores de mutação utilizados por Wong *et al.* A descrição de cada um dos operadores de mutação das Tabelas 3.25 e 3.26 está no Apêndice A.

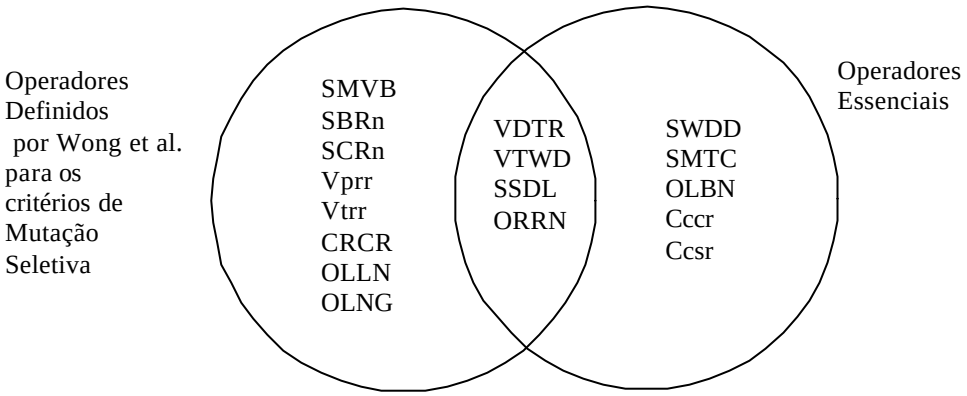

Figura 3.1 – Relação entre os Conjuntos de Operadores de Mutação.

Tabela 3.25 – Conjunto de Operadores Essenciais e os Mutantes Gerados para cada Operador.

Operadores de Mutação	**Mutantes Gerados**
SWDD	37
SMTC	53
SSDL	3848
OLBN	153
ORRN	2765
VTWD	2822
VDTR	4233
Cccr	37504
Ccsr	4077
Total	**55.492**

Tabela 3.26 –Total de Mutantes gerados para o Conjunto de Operadores de Mutação definidos por Wong.

Operadores de Mutação	Mutantes Gerados
VDTR	4233
VTWD	2822
SSDL	3848
SMVB	96
SBRn	0
SCRn	0
Vprr	4210
Vtrr	0
CRCR	5655
OLLN	51
OLNG	153
ORRN	2765
Total	**23.833**

Passo 3 – Aplicação da Técnica baseada em Mutação Seletiva no Programa SPACE

Seguindo os passos realizados por Wong *et al.* (1997a), vide Figura 2.3, e utilizando-se a ferramenta *PROTEUM*, uma sessão de teste foi realizada com o programa SPACE. Após a geração dos mutantes, cada caso de teste t do conjunto T original foi executado um a um. Se esse caso de teste t melhorasse o escore de mutação, então esse era incluído no conjunto T'. Esse procedimento prosseguiu até que todos os casos de teste fossem executados. Ao final do processo, o conjunto T' $\subset$ T AM_adequado foi selecionado.

Ao final do processo de execução dos mutantes, nem todos os mutantes foram distingüidos pelo conjunto de casos de teste disponível. Devido à complexidade do programa SPACE e do alto custo em analisar esses mutantes, todos foram considerados e marcados como equivalentes. Essa, certamente, não é uma abordagem precisa para determinação dos mutantes equivalentes, porém é conservadora. Pode ocorrer que mutantes que não são equivalentes sejam considerados como tal. Como conseqüência, durante a construção do conjunto AM_adequado, pode-se deixar de selecionar um caso de teste que distinguiria aquele mutante. Isso, por sua vez, só pode fazer com o que o conjunto AM_adequado seja menos eficaz em revelar um defeito, mas nunca mais eficaz. Dessa forma, as medidas de eficácia são pessimistas (Delamaro, 1997a).

Ao final do processo descrito acima, o conjunto T' definido para o programa SPACE com relação aos operadores essenciais de mutação possui 229 casos de teste. No entanto, como se pode notar pela Tabela 3.24, nenhum caso de teste do conjunto T original é capaz de revelar o

defeito D8. Assim, um novo caso de teste foi selecionado, de um outro conjunto de casos de teste do programa SPACE, para o teste de regressão,.

Passo 4 - Análise dos Resultados obtidos

A Tabela 3.27 mostra os resultados de redução de tamanho, eficácia, precisão e *recall* obtidos com os 229 casos de teste selecionados pelo processo descrito acima com relação ao conjunto de casos de teste original T, e a Tabela 3.28 mostra os resultados da precisão e do *recall* do conjunto de teste de regressão T' com relação a cada defeito.

Analisando a Tabela 3.27, nota-se que não houve redução de eficácia, pois o conjunto T original não revela o defeito D8, como já mencionado. A redução de tamanho foi de 77,100%, enquanto que o *recall* foi de 27,126%. Dos 229 casos de teste que foram selecionados, 95 não revelam defeitos. Sendo assim, a precisão do conjunto de teste de regressão T' é relativamente baixa, 58,515%, ou seja, 134 casos de teste revelam algum defeito.

Analisando a Tabela 3.28, observa-se que os valores de *recall* foram relativamente baixos, média de 36,281%, sendo maior para o defeito D9, 62,069%, e menor para o defeito D10, 25,245%. O mesmo ocorre com a precisão de T' com relação a cada um dos defeitos, com média de 8,626, sendo maior para o defeito D10, 44,978%, e o menor para os defeitos D4 e D5, 0,437%. Nota-se pela Tabela 3.24 que os defeitos que possuem um número maior de casos de teste que o revelam são aqueles com os maiores valores de precisão.

Tabela 3.27 – Redução de Tamanho, Eficácia, Precisão e Recall do conjunto T' selecionado com relação ao conjunto T original.

Redução de Tamanho	Eficácia	Precisão	*Recall*
77,100	90	58,515	27,126

Tabela 3.28- Precisão e Recall do conjunto T' com relação a cada defeito.

Defeito	Precisão	*Recall*
D1	3,057	38,889
D2	4,637	55,556
D3	11,790	25,472
D4	0,437	33,333
D5	0,437	33,333
D6	2,183	26,316
D7	2,183	26,316
D8	-	-
D9	7,860	62,069
D10	44,978	25,245

3.2.3 Análise Comparativa dos Resultados obtidos com a aplicação de ambas as Técnicas de Teste de Regressão com os resultados obtidos por Wong *et al.* (1997a, 1997b)

Os resultados obtidos nos experimentos realizados foram avaliados sob duas perspectivas: 1 - avaliou-se a técnica de teste de regressão considerando seu comportamento com relação a diferentes complexidades de programas; e 2 - avaliou-se como diferentes técnicas de teste de regressão se comportam em determinados domínios ou conjuntos de programas.

Técnica baseada em Modificação

Os resultados apresentados para os experimentos de Wong *et al.*(1997a) estão sintetizados na Tabela 3.29. Para o experimento conduzido por Wong, descrito no capítulo anterior, utilizando a técnica baseada em Modificação no programa SPACE, em média, os conjuntos baseados em modificação T" continham 80,3 casos de teste *modification-traversing*, sendo o menor com quatro casos de teste e o maior com 4 70 casos de teste. Foram definidos dez conjuntos de casos de teste *modification-traversing*, um para cada modificação do programa SPACE. Após aplicar os mecanismos de minimização e priorização, em média, os conjuntos minimizados continham 9,9 casos de teste, enquanto que os conjuntos priorizados continham, no primeiro conjunto, 3,4 casos de teste, no segundo conjunto 6,8 casos de teste e para no terceiro conjunto 9,9 casos de teste. Aplicando o mecanismo de priorização, Wong selecionou três conjuntos priorizados para cada um dos defeitos.

Analisando-se a Tabela 3.29, nota-se que os conjuntos minimizados e os conjuntos priorizados 3 obtiveram os mesmos valores de redução de tamanho, 70,917%, enquanto que os conjuntos priorizados 1 e 2 obtiveram valores maiores de redução de tamanho, 88,552% e 77,103%, respectivamente. Os valores de precisão foram semelhantes para todos conjuntos minimizados e priorizados, apresentando apenas uma pequena variação nos valores. Os conjuntos minimizados e os conjuntos priorizados 3 obtiveram valores semelhantes de *recall*, enquanto que os conjuntos priorizados 1 e 2 obtiveram menores valores.

Para os experimentos conduzidos durante este trabalho, uma síntese dos resultados é apresentada nas Tabelas 3.30 e 3.31. Nota-se pelos resultados apresentados nas Tabelas 3.29, 3.30 e 3.31, que a técnica baseada em modificação foi capaz de reduzir bastante o número de casos de teste a serem reexecutados tanto para programas de grande porte, como o SPACE, quanto para programas pequenos, como os programas utilitários do UNIX. Para todos os

programas, não houve redução de eficácia dos conjuntos de casos de teste de regressão selecionados com relação ao conjunto de casos de teste T original.

Comparando os resultados com os diferentes programas, nota-se que diferentes complexidades dos programas não afetaram o desempenho da técnica quanto à eficácia. Os valores de precisão dos programas Look, Uniq e SPACE para os conjuntos minimizados foram semelhantes, enquanto que para os programas Cal, Checkeq e Comm esses valores foram significativamente menores que do programa SPACE. Com relação aos conjuntos priorizados, os programas Comm e SPACE tiveram valores semelhantes, e os programas Cal, Checkeq, Look e Uniq tiveram valores significativamente maiores.

Tabela 3.29 – Média para os conjuntos minimizados e priorizados do programa SPACE.

Conjunto	Redução de Tamanho	Eficácia	Precisão	*Recall*
Minimizados	70,917	100	71,763	35,9
Priorizados_1	88,552	100	69,001	12,038
Priorizados_2	77,103	100	68,083	22,561
Priorizados_3	70,917	100	70,254	34,742

Tabela 3.30 – Média para os conjuntos minimizados $T"_{MDx}$ para os programa utilitários do UNIX.

Programas	Redução de Tamanho	Eficácia	Precisão	Recall
Cal	89,712	100	59,815	16,293
Checkeq	92,251	95,455	43,613	10,592
Comm	97,419	100	48,636	9,737
Look	90,343	100	73,315	27,243
Uniq	98,831	100	78,191	1,838

Tabela 3.31 – Média para os conjuntos priorizados $T"_{PDx}$ para os programa utilitários do UNIX.

Programas	Redução de Tamanho	Eficácia	Precisão	Recall
Cal	90,200	100	90,625	18,159
Checkeq	93,164	95,455	91,270	9,828
Comm	99,046	100	72,778	13,740
Look	99,142	100	93,750	21,435
Uniq	98,377	100	88,889	3,723

Técnica baseada em Mutação Seletiva

Para o experimento conduzido por Wong *et al.* (1997b), descrito no capítulo anterior, utilizando a técnica baseada em Mutação Seletiva nos programas utilitários do UNIX, todos os conjuntos de teste de regressão reduzidos foram capazes de revelar os defeitos com uma grande redução do número de casos de teste a serem reexecutados. Ex.: Para o programa Cal, em média, cada conjunto de teste de regressão reduzido continha 17,4 casos de teste e a eficácia foi de 100% para todos os conjuntos com relação aos defeitos disponíveis (Tabela 3.32). Os valores de

precisão e *recall* não foram apresentados por Wong *et al.* assim como os resultados para cada um dos demais programas utilitários do UNIX.

Tabela 3.32 – Número de casos de teste selecionados para os conjuntos de teste de regressão para o programa Cal.

Número de casos de teste selecionados	Número de defeitos revelados
20	20
16	20
17	20
16	20
18	20

Aplicando a técnica baseada em Mutação Seletiva no programa SPACE, muitos casos de teste foram selecionados para serem reexecutados. Isso ocorreu devido à complexidade do programa SPACE. O conjunto de casos de teste selecionado para o programa SPACE continha 229 casos de teste, uma redução de 77,100% com relação ao conjunto T original (1000 casos de teste). Para o programa Cal, aplicando a mesma técnica, obteve-se, em média, 17,4 casos de teste nos conjuntos reduzidos, com uma redução média de tamanho de 89,259% com relação ao conjunto T original (162 casos de teste).

Em termos de porcentagem, a diferença de redução de tamanho não foi grande, 12,159%, evidenciando que a técnica foi capaz de reduzir significativamente o tamanho dos conjuntos. Assim, observa-se que a técnica é sensível à complexidade do programa. No entanto, a técnica foi capaz de reduzir o conjunto de teste de regressão para um número de casos de teste aceitável para as atividades de teste de regressão, mesmo para programas mais complexos. A eficácia dos conjuntos em revelar defeitos não foi afetada pelas diferentes complexidades dos programas.

Notou-se que a técnica baseada em Mutação Seletiva é mais sensível à complexidade dos programas em teste do que a técnica baseada em Modificação, pois esta seleciona casos de teste a partir das modificações realizadas e aquela a partir da Análise de Mutantes.

3.2.4 Avaliação das Técnicas de Teste de Regressão segundo o *Framework* de Rothermel e Harrold

Nesta seção, as técnicas são avaliadas segundo o *framework* de Rothermel e Harrold. Para avaliar a técnica baseada em Modificação, utilizaram-se os resultados obtidos com os programas utilitários do UNIX, e para avaliar a técnica baseada em Mutação seletiva, utilizaram-se os resultados obtidos com o programa SPACE. Relembrando, as seguintes características são avaliadas pelo *framework*:

- **Inclusão** - mede a extensão com a qual a técnica escolhe casos de teste que fazem com que **P'** produza uma saída diferente de **P** e revelam defeitos, casos de teste *modification-revealing*.
- **Precisão** - mede a habilidade da técnica em omitir casos de teste que não fazem com que **P'** produza uma saída diferente de **P**, casos de teste *non_modification-revealing*.
- **Eficiência** - mede o custo computacional gasto pela técnica.
- **Generalidade** - mede a habilidade da técnica em manipular linguagens diversas.

Observa-se que a precisão medida pelo *framework* se difere da precisão medida na seção anterior, no sentido de que uma calcula a habilidade da técnica em omitir casos de teste *non_modification-revealing*, e a outra calcula a porcentagem de casos de teste do conjunto de teste de regressão selecionado que são *modification-revealing*. Assim, para diferenciar, a precisão definida pelo *framework* é referenciada como precisao_F.

As Tabelas 3.33 e 3.34 sintetizam os resultados obtidos aplicando-se a técnica baseada em Modificação nos programas utilitários do UNIX, e a Tabela 3.35 sintetiza os resultados obtidos com a aplicação da técnica baseada em Mutação Seletiva no programa SPACE.

Com relação aos programas utilitários do UNIX, como se pode observar, os valores de inclusão para os conjuntos minimizados e priorizados foram semelhantes, apresentando uma maior diferença para o programa Look, 5,808%. A precisão_F para os conjuntos minimizados e priorizados apresentou uma maior diferença para o programa Cal, 5%. Os valores de precisão_F para o programa Checkeq não foram calculados, pois para esse programa o conjunto T original de casos de teste não possui casos de teste *non_modification-revealing*.

Para todos os programas utilitários do UNIX, observou-se que o mecanismo de priorização utilizado foi capaz de omitir todos os casos de teste *non_modification-revealing*. Já o mecanismo de minimização não foi capaz de omitir todos esses casos de teste. No entanto, mesmo não utilizando informações sobre a eficácia dos casos de teste, o mecanismo de minimização foi capaz de omitir a maioria dos casos de teste *non_modification-revealing*. Exceto para o programa Cal, a média de omissão desses casos de teste foi superior a 98%.

Com relação ao programa SPACE, o conjunto T original disponível para esse programa possui muitos casos de teste *non_modification-revealing*, 50,6% dos casos, que equivale a 506 casos de teste. Mesmo assim, a técnica baseada em Mutação Seletiva foi capaz de evitar 81,225% desses casos de teste. A inclusão para o conjunto do programa SPACE foi de 27,216% com relação ao conjunto T original.

Tabela 3.33 – Técnica baseada em Modificação: Média dos resultados de inclusão e precisão_F para os conjuntos minimizados dos programas utilitários do UNIX.

Programas	Inclusão	Precisão_F
Cal	16,293	95
Checkeq	10,592	---
Comm	9,737	98,148
Look	27,243	99,167
Uniq	1,838	99,647

Tabela 3.34 – Técnica baseada em Modificação: média dos resultados de inclusão e precisão_F para os conjuntos priorizados dos programas utilitários do UNIX.

Programas	Inclusão	Precisão_F
Cal	18,159	100
Checkeq	9,828	---
Comm	13,740	100
Look	21,435	100
Uniq	3,723	100

Tabela 3.35 – Técnica baseada em Mutação Seletiva: resultados de inclusão e precisão_F para o programa SPACE.

Programa	Inclusão	Precisão_F
SPACE	27,126	81,225

A seguir são apresentados os resultados com relação a cada um dos programas para cada uma das técnicas.

➔ **Técnica baseada em Modificação**

Inclusão e **Precisão_F**

Os valores de inclusão para os conjuntos minimizados e priorizados dos programas utilitários do UNIX são apresentados nas Tabelas 3.36 e 3.37, respectivamente.

Para o programa Cal, a precisão_F para os conjuntos minimizados foi de 66,667% para os conjuntos T"$_{MD1}$, T"$_{MD5}$ e T"$_{MD9}$ e de 100% para os demais conjuntos. Assim, 85% dos conjuntos não possuem casos de teste *non_modification-revealing*. A precisão_F para os conjuntos priorizados foi de 100% para todos os defeitos. Assim, todos os conjuntos não possuem casos de teste *non_modification-revealing*. Na média, para os conjuntos minimizados a inclusão foi de 16,293% e para os conjuntos priorizados foi de 18,159%.

Como para o programa Checkeq todos os casos de teste disponíveis no conjunto original são capazes de revelar pelo menos um defeito do conjunto utilizado, a precisão_F não foi

calculada para os conjuntos minimizados e priorizados. A inclusão dos conjuntos minimizados, em média, foi de 10,592%, e 9,828%, em média, para os conjuntos priorizados.

Para o programa Comm, a média de inclusão para os conjuntos minimizados foi de 9,737%, e 13,740% para os conjuntos priorizados. Já a precisão_F foi de 100% para os conjuntos priorizados e 98,148%, em média, para os conjuntos minimizados. Nesse caso, somente para o defeito D1, a técnica não foi capaz de omitir todos os casos *non_modification-revealing*, sendo a precisão_F para esse conjunto de 66,667%.

Para o programa Look, a inclusão obteve uma média de 27,243% para os conjuntos minimizados e 21,435% para os conjuntos priorizados. Para todos os conjuntos priorizados, a precisão_F foi de 100%, enquanto que para os conjuntos minimizados, essa média foi de 99,167%. Apenas para os defeitos D1 e D18, os conjuntos minimizados não obtiveram uma precisão_F de 100%, e sim de 93,333%.

O Uniq foi o programa que obteve uma menor média de inclusão com relação aos outros quatro programas. Para os conjuntos minimizados, a média foi de 1,838%, e para os conjuntos priorizados a média foi de 3,723%. A precisão_F para todos conjuntos priorizados foi de 100%. Já para os conjuntos minimizados, a média foi de 99,647%. Para os defeitos D1, D12 e D15, a precisão_F foi de 99,091%, e para os defeitos D7 e D8, a precisão_F foi de 98,182%. Para os demais defeitos, a precisão_F foi de 100%.

O programa Uniq, vide Tabela 3.3, é o programa que possui o maior número de casos de teste *non_modification-revealing*, 110 casos de teste. Mesmo assim, o programa apresentou uma maior precisão_F para os conjuntos minimizados com relação aos outros programas utilitários do UNIX.

Eficiência: utilizou-se a ferramenta *PROTEUM/IM* para estabelecer o *trace* de execução de um caso de teste, não comprometendo, assim, o custo de aplicação da técnica. O esforço maior foi despendido na tarefa de identificar em quais blocos de comandos as modificações foram realizadas. Essa tarefa foi realizada analisando-se o grafo de fluxo de controle dos programas originais. Comparando o *trace* de execução com o grafo de fluxo de controle, foi possível definir os conjuntos de casos de teste *modification-traversing* para cada modificação. Essas tarefas foram realizadas utilizando-se *scripts* escritos no C-*shell* do UNIX.

Generalidade: a técnica não depende da linguagem do programa em teste, nem mesmo da técnica aplicada para o teste. Independentemente das modificações realizadas, a técnica é capaz de selecionar um conjunto de casos de teste que as exercitem, sejam elas de remoção, alteração ou inserção de comandos, intra ou interprocedurais. Assim, qualquer ferramenta de

teste que seja capaz de fornecer o *trace* de execução de um caso de teste pode apoiá-la computacionalmente, seja uma ferramenta para o teste estrutural ou para o teste de mutação.

Tabela 3.36 – Inclusão para os conjuntos minimizados dos programas utilitários do UNIX.

Defeitos\Programas	Cal	Checkeq	Comm	Look	Uniq
$T"_{MD1}$	7,971	27,273	3,149	100	2,308
$T"_{MD2}$	100	5,298	33,333	7,650	1,220
$T"_{MD3}$	9,574	6,040	-	50	1,010
$T"_{MD4}$	12,360	2,970	3,846	-	0,296
$T"_{MD5}$	40	10,891	3,170	13,043	1,124
$T"_{MD6}$	18,421	-	8,333	16,129	1,176
$T"_{MD7}$	8,661	10,891	8,333	20,588	0,763
$T"_{MD8}$	8,661	3,623	2,575	17,949	0,847
$T"_{MD9}$	12,195	2,190	1,515	-	3,077
$T"_{MD10}$	7,477	6,289	33,333	-	1,389
$T"_{MD11}$	11,290	13,333	2,970	25,714	-
$T"_{MD12}$	7,778	3,922	0,800	33,333	0,704
$T"_{MD13}$	8,661	81,818	1,613	22,222	0,690
$T"_{MD14}$	8,661	3,571	3,030	21,622	3,333
$T"_{MD15}$	7,759	6,306	4,104	-	6,667
$T"_{MD16}$	8,661	4,255	2,532	10,526	1,282
$T"_{MD17}$	8,943	9,195	33,333	5,442	0,685
$T"_{MD18}$	13,793	3,401	2,016	50	4,762
$T"_{MD19}$	9,375	2,353	27,273	25	1,754
$T"_{MD20}$	15,625	13,514	-	16,667	-
$T"_{MD21}$	-	2,899	-		-
$T"_{MD22}$	-	2,410	-		-

Tabela 3.37 – Inclusão para os conjuntos priorizados dos programas utilitários do UNIX.

Defeitos\Programas	Cal	Checkeq	Comm	Look	Uniq
$T"_{PD1}$	3,623	3,030	0,504	100	1,923
$T"_{PD2}$	100	7,285	33,333	0,546	3,659
$T"_{PD3}$	5,319	7,383	-	50	2,020
$T"_{PD4}$	5,618	10,891	3,846	-	1,479
$T"_{PD5}$	20,000	5,446	1,153	4,348	1,124
$T"_{PD6}$	18,421	-	75	3,226	2,353
$T"_{PD7}$	9,449	5,446	16,667	2,941	3,053
$T"_{PD8}$	9,449	7,971	1,717	2,564	2,119
$T"_{PD9}$	21,951	8,029	0,866	-	7,692
$T"_{PD10}$	11,215	6,918	33,333	-	4,167
$T"_{PD11}$	19,355	24,444	0,792	2,857	-
$T"_{PD12}$	13,333	7,190	0,800	33,333	3,521
$T"_{PD13}$	9,449	9,091	3,226	2,778	3,448
$T"_{PD14}$	9,449	7,857	2,020	2,703	8,333
$T"_{PD15}$	10,345	9,910	1,493	-	8,333
$T"_{PD16}$	9,449	7,801	1,266	0,877	3,846
$T"_{PD17}$	9,756	6,322	33,333	0,680	3,425
$T"_{PD18}$	37,931	7,483	1,613	50	4,762
$T"_{PD19}$	17,188	12,941	36,364	2,778	1,754
$T"_{PD20}$	21,875	29,730	-	83,333	-
$T"_{PD21}$	-	7,971	-	-	-
$T"_{PD22}$	-	13,253	-	-	-

➔ **Técnica baseada em Mutação Seletiva**

Inclusão e **Precisão_F**

A precisão_F para o conjunto de teste de regressão selecionado foi de 81,225%, ou seja, 411 dos 506 casos de teste *non_modification-revealing* foram omitidos. Mesmo não tendo informações sobre a eficácia dos casos de teste disponíveis, a técnica foi capaz de omitir a maioria dos casos de teste *non_modification-revealing*. A inclusão para o conjunto selecionado com relação ao conjunto original foi de 27,126%. Os valores da inclusão para cada defeito estão na Tabela 3.38, e foram, em média, de 36,281%.

Tabela 3.38 – Inclusão para os defeitos do programa SPACE.

Defeito	Inclusão
D1	38,889
D2	55,556
D3	25,472
D4	33,333
D5	33,333
D6	26,316
D7	26,316
D8	-
D9	62,069
D10	25,245

Eficiência: como o programa utilizado é de grande porte, a aplicação da técnica demandou um alto custo computacional, pois muitos mutantes foram gerados pela ferramenta *PROTEUM* mesmo utilizando-se um subconjunto de operadores de mutação. No entanto, a técnica foi capaz de reduzir o conjunto de teste de regressão para um número de casos de teste aceitável.

Generalidade: essa técnica também não depende da linguagem do programa em teste, no entanto, depende do critério de teste utilizado, no caso a Mutação Seletiva. Assim, para a aplicação computacional dessa técnica é necessário ter disponível uma ferramenta de teste que apóie a mutação seletiva. Além disso, é necessário também ter um prévio conhecimento do critério de teste e dos conceitos de teste de mutação.

3.3 Considerações Finais

Neste capítulo foram apresentados os experimentos realizados para avaliar a aplicação de duas técnicas de teste de regressão: Técnica baseada em Modificação e Técnica baseada em Mutação Seletiva.

Observa-se que a técnica baseada em Mutação Seletiva é mais sensível à complexidade do programa em teste do que a técnica baseada em Modificação. No entanto, mesmo sendo mais sensível à complexidade, a técnica foi capaz de reduzir o conjunto de teste de regressão para um número aceitável de casos de teste a serem reexecutados. Observou-se também que, dependendo do mecanismo de minimização ou priorização aplicado, é possível deixar uma técnica mais segura, garantindo que casos de teste *non_modification-revealing* não sejam selecionados. Ambas as técnicas foram capazes de selecionar um conjunto reduzido de casos de teste para o teste de regressão sem afetar a eficácia do conjunto em revelar possíveis defeitos.

Observa-se também que mecanismos de minimização e priorização podem ser aplicados a qualquer técnica de teste de regressão, pois ambos não dependem da técnica utilizada. Rothermel *et al.* (1999) descrevem diversos experimentos utilizando diferentes mecanismos de priorização para auxiliar a seleção de conjuntos de casos de teste de regressão. Neste trabalho utilizou-se a eficácia dos casos de teste disponíveis para estabelecer a prioridade de seleção dos conjuntos de casos de teste de regressão.

Um fato que deve ser ressaltado é que mesmo não utilizando informações sobre a eficácia dos casos de teste, a técnica baseada em Mutação Seletiva foi capaz de reduzir significativamente o conjunto de casos de teste a ser reexecutado sem afetar a eficácia do conjunto em revelar defeitos.

Capítulo 4

Conclusões e Trabalhos Futuros

Ressaltou-se neste trabalho a importância das atividades de teste na fase de manutenção de um software. Outro ponto importante também pertinente para o teste de regressão é a redução de custo de sua aplicação. Como se demonstrou por meio dos experimentos realizados, o teste de regressão é tão caro quanto o teste de software realizado durante o processo de desenvolvimento. Assim, diversas técnicas são propostas na literatura para reduzir os custos do teste de regressão. No entanto, a diversidade de técnicas de teste de regressão dificulta a escolha de uma delas. Diversos estudos empíricos são realizados para avaliar e comparar essas técnicas com o intuito de estabelecer estratégias de teste de regressão menos onerosas para os testadores e facilitar a escolha de uma delas.

Com base nos experimentos realizados por Wong *et al.* (1997a, 1997b) para avaliar as técnicas de teste de regressão baseada em Modificação e Mutação Seletiva com o programa SPACE e os programas utilitários do UNIX, respectivamente, este trabalho avaliou e comparou empiricamente essas duas técnicas de teste de regressão, utilizando a técnica baseada em Modificação nos programas utilitários do UNIX e a técnica baseada em Mutação Seletiva no programa SPACE.

Observou-se com a realização desses experimentos que ambas as técnicas se mostraram eficazes em revelar os defeitos nos programas modificados com um número bastante reduzido de casos de teste a serem reexecutados. A técnica baseada em Mutação Seletiva mostrou-se mais sensível à complexidade do programa em teste do que a técnica baseada em Modificação. O fato da técnica baseada em Mutação Seletiva utilizar conceitos do critério Análise de Mutantes ressalta a sensibilidade da técnica com relação à complexidade, pois programas mais complexos geram mais mutantes e, conseqüentemente, necessitam de mais casos de teste. No entanto, mesmo sendo mais sensível à complexidade dos programas, a técnica foi capaz de reduzir o conjunto de teste de regressão para um número aceitável de casos de teste.

Outro ponto importante é que tanto a técnica baseada em Mutação Seletiva quanto o mecanismo de minimização não se utilizam de informações sobre a eficácia dos casos de teste ao contrário do mecanismo de priorização. No entanto, mesmo assim, ambos foram capazes de reduzir significativamente o número de casos de teste a serem reexecutados no teste de regressão.

Com a aplicação das técnicas descritas neste trabalho, foi possível concluir que técnicas que selecionam casos de teste que exercitam modificações, como é o caso da técnica baseada em Modificação, proporcionam uma maior redução no conjunto de casos de teste original. Estudos realizados por Rothermel e Harrold (1997) e Wong *et al.* (1997b) comprovam que técnicas que se baseiam na cobertura dos componentes modificados são mais eficazes em revelar defeitos. Mesmo não utilizando essas informações de cobertura dos componentes modificados, a técnica baseada em Mutação Seletiva obteve bons resultados de redução de tamanho do conjunto de casos de teste sem afetar a eficácia desse conjunto em revelar defeitos.

Outro ponto importante é que mecanismos de priorização são eficazes em revelar defeitos reduzindo bastante o número de casos de teste a serem reexecutados. Estudos empíricos realizados por Rothermel *et al.* (1999) com diferentes mecanismos de priorização demonstram a eficácia desses mecanismos em revelar defeitos com um número bastante reduzido de casos de teste. Segundo esses estudos, o mecanismo de priorização mais eficaz é aquele que seleciona os casos de teste que mais revelam defeitos, sendo esse o mesmo mecanismo utilizado neste trabalho para aplicar a técnica baseada em Modificação. É importante ressaltar também que tanto mecanismos de priorização quanto mecanismos de minimização independem da técnica de teste de regressão utilizada ou da complexidade dos programas em teste, ou seja, esses mecanismos podem ser incorporados a qualquer técnica de teste de regressão.

4.1 Trabalhos Futuros

Para ilustrar possíveis trabalhos futuros pode-se citar:

- Ampliar o universo de programas utilizados para a comparação das técnicas.
- Avaliar o custo de aplicação das técnicas de acordo com o modelo de custo proposto por Leung e White (1991)

- Utilização de outros mecanismos de priorização para a Técnica baseada em Modificação. Como já mencionado, Rothermel *et al.* (1999) avaliam empiricamente diversos mecanismos de priorização. Alguns desses mecanismos poderiam ser utilizados com o intuito de melhorar a aplicação da Técnica baseada em Modificação e complementar os estudos realizados por Rothermel.
- Utilização do conjunto de operadores essenciais de mutação para o Critério Mutação de Interface, definido por Vincenzi em seu trabalho (1998), para a Técnica baseada em Mutação Seletiva. Assim, a invés de avaliar o quanto que um conjunto de casos de teste AM_adequado é adequado ao programa original e também adequado às suas versões modificadas, avaliar-se-ia o quanto que um conjunto de casos de teste adequado ao critério Mutação de Interface com relação ao programa original é também adequado às suas modificações.
- Explorar essas técnicas no contexto de reengenharia de software.

Apêndice A

Operadores de Mutação

A seguir os operadores de mutação utilizados por Wong *et al.* (1997b) em seus experimentos (Quadro A.1) e os operadores essenciais de mutação utilizados neste trabalho (Quadro A.2) são descritos.

Quadro A.1 – Operadores de Mutação utilizados por Wong et al..

Operador	Descrição
VDTR	Atribui os valores negativo, positivo e zero para cada referência escalar
VTWD	Substitui referência escalar por seu valor antecessor e sucessor
SSDL	Retira um comando de cada vez do programa
SMVB	Move '}' para cima e para baixo quando possível
SBRn	Troca o comando **continue** ou **break** por uma função **break_out_of_level_n(J)** sendo que J varia de acordo com o número de laços aninhados. Essa função força a interrupção dos J laços externos
SCRn	Troca o comando **continue** ou **break** por uma função **continue_out_of_level_n(J)** sendo que J varia de acordo com o número de laços aninhados. Essa função força a transferência do programa para o final de J laços acima
Vprr	Substitui as referências a apontadores por variáveis escalares, globais e locais do programa
Vtrr	Substitui as referências a estruturas e uniões por variáveis escalares, globais e locais do programa
CRCR	Troca constantes por: 0, 1 , -1, dependendo do tipo de referência
OLLN	Troca operador lógico por outro operador lógico
OLNG	Insere negação lógica em condições compostas
ORRN	Troca operador relacional por outro operador relacional

Quadro A.2 –Operadores Essenciais de Mutação.

Operador	Descrição
SWDD	Troca o comando **while** por **do while**
SMTC	Interrompe a execução do laço após 2 execuções
SSDL	Descrito acima
OLBN	Troca operador lógico por operador *bitwise*
ORRN	Descrito acima
VTWD	Descrito acima
VDTR	Descrito acima
Cccr	Troca constantes por outra constante
Ccsr	Troca referências escalares por constantes

Apêndice B

Versões Modificadas dos Programas Utilitários do UNIX e Ativação dos erros do Programa SPACE

Neste Apêndice encontram-se as versões modificadas geradas para os programas utilitários do UNIX utilizados nos experimentos e o processo de ativação de erros do programa SPACE. Utilizou-se a primitiva ***diff*** do UNIX para diferenciar a versão original de cada programa de suas versões. A primitiva ***diff*** tem como entrada, além de parâmetros, os arquivos (diretórios) que devem ser comparados. Após compará-los, a primitiva retorna a linha na qual os arquivos (diretórios) são diferentes e o conteúdo dessas linhas. A primitiva possui a seguinte sintaxe:

diff **[conjunto de parâmetros] arq1 arq 2 / [conjunto de parâmetros] dir 1 dir2**

Versões Modificadas dos Programas utilitários do UNIX

Programa CAL

```
#ifndef lint
static char sccsid[] = "@(#)cal.c   4.4 (Berkeley) 87/05/28";
#endif

#include <sys/types.h>
#include <time.h>
#include <stdio.h>

char  dayw[] = {
      " S  M Tu  W Th  F  S"
};
char  *smon[]= {
      "January", "February", "March", "April",
      "May", "June", "July", "August",
      "September", "October", "November", "December",
};
```

```
char  string[432];
main(argc, argv)
char *argv[];
{
      register y, i, j;
      int m;

      if(argc == 2)
            goto xlong;
      /*
       * print out just month
       */
      if(argc < 2) {                /* current month */
            time_t t;
            struct tm *tm;

            t = time(0);
            tm = localtime(&t);
            m = tm->tm_mon + 1;
            y = tm->tm_year + 1900;
      } else {
            m = atoi(argv[1]);
            if(m<1 || m>12) {
                  fprintf(stderr, "cal: %s: Bad month.\n", argv[1]);
                  exit(1);
            }
            y = atoi(argv[2]);
            if(y<1 || y>9999) {
                  fprintf(stderr, "cal: %s: Bad year.\n", argv[2]);
                  exit(2);
            }
      }
      printf("   %s %u\n", smon[m-1], y);
      printf("%s\n", dayw);
      cal(m, y, string, 24);
      for(i=0; i<6*24; i+=24)
            pstr(string+i, 24);
      exit(0);

xlong:
      /*
       * print out complete year
       */
      y = atoi(argv[1]);
      if(y<1 || y>9999) {
            fprintf(stderr, "cal: %s: Bad year.\n", argv[1]);
            exit(2);
      }
      printf("\n\n\n");
      printf("                      %u\n", y);
      printf("\n");
      for(i=0; i<12; i+=3) {
            for(j=0; j<6*72; j++)
                  string[j] = '\0';
            printf("     %.3s", smon[i]);
            printf("                %.3s", smon[i+1]);
            printf("                 %.3s\n", smon[i+2]);
            printf("%s   %s   %s\n", dayw, dayw, dayw);
            cal(i+1, y, string, 72);
            cal(i+2, y, string+23, 72);
```

```
            cal(i+3, y, string+46, 72);
            for(j=0; j<6*72; j+=72)
                  pstr(string+j, 72);
      }
      printf("\n\n\n");
      exit(0);
}

pstr(str, n)
char *str;
{
      register i;
      register char *s;

      s = str;
      i = n;
      while(i--)
            if(*s++ == '\0')
                  s[-1] = ' ';
      i = n+1;
      while(i--)
            if(*--s != ' ')
                  break;
      s[1] = '\0';
      printf("%s\n", str);
}

char  mon[] = {
      0,
      31, 29, 31, 30,
      31, 30, 31, 31,
      30, 31, 30, 31,
};

cal(m, y, p, w)
char *p;
{
      register d, i;
      register char *s;

      s = p;
      d = jan1(y);
      mon[2] = 29;
      mon[9] = 30;

      switch((jan1(y+1)+7-d)%7) {

      /*
       *    non-leap year
       */
      case 1:
            mon[2] = 28;
            break;

      /*
       *    1752
       */
      default:
            mon[9] = 19;
            break;
```

```
	/*
	 *	leap year
	 */
	case 2:
		;
	}
	for(i=1; i<m; i++)
		d += mon[i];
	d %= 7;
	s += 3*d;
	for(i=1; i<=mon[m]; i++) {
		if(i==3 && mon[m]==19) {
			i += 11;
			mon[m] += 11;
		}
		if(i > 9)
			*s = i/10+'0';
		s++;
		*s++ = i%10+'0';
		s++;
		if(++d == 7) {
			d = 0;
			s = p+w;
			p = s;
		}
	}
}

/*
 *	return day of the week
 *	of jan 1 of given year
 */

jan1(yr)
{
	register y, d;

/*
 *	normal gregorian calendar
 *	one extra day per four years
 */

	y = yr;
	d = 4+y+(y+3)/4;

/*
 *	julian calendar
 *	regular gregorian
 *	less three days per 400
 */

	if(y > 1800) {
		d -= (y-1701)/100;
		d += (y-1601)/400;
	}

/*
 *	great calendar changeover instant
 */
```

```
	if(y > 1752)
		d += 3;

	return(d%7);
}
```

Versões Do Programa CAL

```
=====diff Cal.c and Cal-1.c================
24c24
<     if(argc == 2)
---
>     if(argc = 2)
=====diff Cal.c and Cal-2.c================
36c36
<           y = tm->tm_year + 1900;
---
>           y = tm->tm_year + 1800;
=====diff Cal.c and Cal-3.c================
44c44
<           if(y<1 || y>9999) {
---
>           if(y<1 || y<9999) {
=====diff Cal.c and Cal-4.c================
52c52
<     for(i=0; i<6*24; i+=24)
---
>     for(i=0; i<=6*24; i+=24)
=====diff Cal.c and Cal-5.c================
61c61
<     if(y<1 || y>9999) {
---
>     if(y>9999) {
=====diff Cal.c and Cal-6.c================
68c68
<     for(i=0; i<12; i+=3) {
---
>     for(i=0; i<12; i+=4) {
=====diff Cal.c and Cal-7.c================
94c94
<           if(*s++ == '\0')
---
>           if(*++s == '\0')
=====diff Cal.c and Cal-8.c================
98c98
<           if(*--s != ' ')
---
>           if(*--s == ' ')
=====diff Cal.c and Cal-9.c================
108c108
<     30, 31, 30, 31,
---
>     30, 31, 30, 30,
=====diff Cal.c and Cal-10.c===============
119c119
<     mon[2] = 29;
```

```
---
>     mon[3] = 29;
=====diff Cal.c and Cal-11.c================
120c120
<     mon[9] = 30;
---
>     mon[9] = 31;
=====diff Cal.c and Cal-12.c================
127,129d126
<     case 1:
<          mon[2] = 28;
<          break;
=====diff Cal.c and Cal-13.c================
149c149
<          if(i==3 && mon[m]==19) {
---
>          if(i==3 || mon[m]==19) {
=====diff Cal.c and Cal-14.c================
155d154
<          s++;
=====diff Cal.c and Cal-15.c================
146c146
<     d %= 7;
---
>     i %= 7;
=====diff Cal.c and Cal-16.c================
160c160
<                s = p+w;
---
>                s = p;
=====diff Cal.c and Cal-17.c================
181c181
<     d = 4+y+(y+3)/4;
---
>     d = 4+y+(y*3)/4;
=====diff Cal.c and Cal-18.c================
191c191
<          d += (y-1601)/400;
---
>          d += y-1601/400;
=====diff Cal.c and Cal-19.c================
199c199
<          d += 3;
---
>          d = 3;
=====diff Cal.c and Cal-20.c================
77c77
<          cal(i+3, y, string+46, 72);
---
>          cal(i+3, j, string+46, 72);
```

Programa Checkeq

```
/*
 * Copyright (c) 1987 Regents of the University of California.
 * All rights reserved.  The Berkeley software License Agreement
 * specifies the terms and conditions for redistribution.
 */

#ifndef lint
char copyright[] =
"@(#) Copyright (c) 1987 Regents of the University of California.\n\
 All rights reserved.\n";
#endif /* not lint */

#ifndef lint
static char sccsid[] = "@(#)checkeq.c    4.3 (Berkeley) 12/2/87";
#endif /* not lint */

#include <stdio.h>
FILE  *fin;
int   delim = '$';

main(argc, argv) char **argv; {

      if (argc <= 1)
            check(stdin);
      else
            while (--argc > 0) {
                  if ((fin = fopen(*++argv, "r")) == NULL) {
                        perror(*argv);
                        exit(1);
                  }
                  printf("%s:\n", *argv);
                  check(fin);
                  fclose(fin);
            }
      exit(0);
}

check(f)
FILE  *f;
{
      int start, line, eq, ndel, totdel;
      char in[600], *p;

      start = eq = line = ndel = totdel = 0;
      while (fgets(in, 600, f) != NULL) {
            line++;
            ndel = 0;
            for (p = in; *p; p++)
                  if (*p == delim)
                        ndel++;
            if (*in=='.' && *(in+1)=='E' && *(in+2)=='Q') {
                  if (eq++)
                        printf("   Spurious EQ, line %d\n", line);
                  if (totdel)
                        printf("   EQ in %c%c, line %d\n", delim, delim,
line);
            } else if (*in=='.' && *(in+1)=='E' && *(in+2)=='N') {
                  if (eq==0)
```

```
                        printf("   Spurious EN, line %d\n", line);
                  else
                        eq = 0;
                  if (totdel > 0)
                        printf("   EN in %c%c, line %d\n", delim, delim,
line);
                  start = 0;
            } else if (eq && *in=='d' && *(in+1)=='e' && *(in+2)=='l' &&
*(in+3)=='i' && *(in+4)=='m') {
                  for (p=in+5; *p; p++)
                        if (*p != ' ') {
                              if (*p == 'o' && *(p+1) == 'f')
                                    delim = 0;
                              else
                                    delim = *p;
                              break;
                        }
                  if (delim == 0)
                        printf("   Delim off, line %d\n", line);
                  else
                        printf("   New delims %c%c, line %d\n", delim, delim,
line);
            }
            if (ndel > 0 && eq > 0)
                  printf("   %c%c in EQ, line %d\n", delim, delim, line);
            if (ndel == 0)
                  continue;
            totdel += ndel;
            if (totdel%2) {
                  if (start == 0)
                        start = line;
                  else {
                        printf("   %d line %c%c, lines %d-%d\n", line-start+1,
delim, delim, start, line);
                        start = line;
                  }
            } else {
                  if (start > 0) {
                        printf("   %d line %c%c, lines %d-%d\n", line-start+1,
delim, delim, start, line);
                        start = 0;
                  }
                  totdel = 0;
            }
      }
      if (totdel)
            printf("   Unfinished %c%c\n", delim, delim);
      if (eq)
            printf("   Unfinished EQ\n");
}
```

Versões Do Programa CHECKEQ

```
=====diff Checkeq.c and Checkeq-1.c================
23c23
<     if (argc <= 1)
---
>     if (argc <= 0)
=====diff Checkeq.c and Checkeq-2.c================
26c26
<           while (--argc > 0) {
---
>           while (argc-- > 0) {
=====diff Checkeq.c and Checkeq-3.c================
27c27
<           if ((fin = fopen(*++argv, "r")) == NULL) {
---
>           if ((fin = fopen(*argv++, "r")) == NULL) {
=====diff Checkeq.c and Checkeq-4.c================
93c93
<                       start = 0;
---
>                       totdel = 0;
=====diff Checkeq.c and Checkeq-5.c================
44c44
<     start = eq = line = ndel = totdel = 0;
---
>     start = line = ndel = totdel = 0;
=====diff Checkeq.c and Checkeq-6.c================
45c45
<     while (fgets(in, 600, f) != NULL) {
---
>     while (fgets(in, 300, f) != NULL) {
=====diff Checkeq.c and Checkeq-7.c================
47c47
<           ndel = 0;
---
>           ndel = 1;
=====diff Checkeq.c and Checkeq-8.c================
50c50
<                       ndel++;
---
>                       ndel--;
=====diff Checkeq.c and Checkeq-9.c================
95d94
<                 totdel = 0;
=====diff Checkeq.c and Checkeq-10.c================
51c51
<     if (*in=='.' && *(in+1)=='E' && *(in+2)=='Q') {
---
>     if (*in=='.' && *(in+1)=='E' || *(in+2)=='Q') {
=====diff Checkeq.c and Checkeq-11.c================
56c56
<     } else if (*in=='.' && *(in+1)=='E' && *(in+2)=='N') {
---
>     } else if (*in=='.' && *(in+1)=='E' || *(in+2)=='N') {
=====diff Checkeq.c and Checkeq-12.c================
60c60
<                       eq = 0;
```

```
---
>                       eq = 1;
=====diff Checkeq.c and Checkeq-13.c===============
64c64
<     } else if (eq && *in=='d' && *(in+1)=='e'
          && *(in+2)=='l' && *(in+3)=='i' && *(in+4)=='m') {
---
>           } else if (eq && *in=='d') {
=====diff Checkeq.c and Checkeq-14.c===============
70c70
<                       delim = *p;
---
>                       line = *p;
=====diff Checkeq.c and Checkeq-15.c===============
70c70
<                       delim = *p;
---
>                       delim = *(++p);
=====diff Checkeq.c and Checkeq-16.c===============
75,76d74
<                 else
<     printf("   New delims %c%c, line %d\n", delim, delim, line);
=====diff Checkeq.c and Checkeq-17.c===============
78c78
<           if (ndel > 0 && eq > 0)
---
>           if (ndel > 0 || eq > 0)
=====diff Checkeq.c and Checkeq-18.c===============
78c78
<           if (ndel > 0 && eq > 0)
---
>           if (ndel > 0 && eq <= 0)
=====diff Checkeq.c and Checkeq-19.c===============
82c82
<           totdel += ndel;
---
>           totdel -= ndel;
=====diff Checkeq.c and Checkeq-20.c===============
86,89d85
<                 else {
<     printf("   %d line %c%c, lines %d-%d\n",
                        line-start+1, delim, delim, start, line);
<                       start = line;
<                 }
=====diff Checkeq.c and Checkeq-21.c===============
91c91
<                 if (start > 0) {
---
>                 if (start == 0) {
=====diff Checkeq.c and Checkeq-22.c===============
54c54
<                 if (totdel)
---
>                 if (ndel)
```

Programa Comm

```
static char *sccsid = "@(#)comm.c  4.2 (Berkeley) 4/29/83";
#include <stdio.h>
#define LB 256
int   one;
int   two;
int   three;

/* char     *ldr[3]; */
char  ldr[3][2]={"\0","\t","\t\t"};

FILE *ib1;
FILE *ib2;
FILE *openfil();
main(argc,argv)
char  *argv[];
{
      int   l;
      char  lb1[LB],lb2[LB];
/*
      ldr[0] = "";
      ldr[1] = "\t";
      ldr[2] = "\t\t";
*/

      if(argc > 1)  {
            if(*argv[1] == '-' && argv[1][1] != 0) {
                  l = 1;
                  while(*++argv[1]) {
                        switch(*argv[1]) {
                        case'1':
                              if(!one) {
                                    one = 1;
                                    ldr[1][0] =
                                    ldr[2][l--] = '\0';
                              }
                              break;
                        case '2':
                              if(!two) {
                                    two = 1;
                                    ldr[2][l--] = '\0';
                              }
                              break;
                        case '3':
                              three = 1;
                              break;
                        default:
                        fprintf(stderr,"comm: illegal flag\n");
                        exit(1);
                        }
                  }
                  argv++;
                  argc--;
            }
      }

      if(argc < 3) {
            fprintf(stderr,"comm: arg count\n");
            exit(1);
```

```
	}

	ib1 = openfil(argv[1]);
	ib2 = openfil(argv[2]);

	if(rd(ib1,lb1) < 0) {
		if(rd(ib2,lb2) < 0)	exit(0);
		copy(ib2,lb2,2);
	}
	if(rd(ib2,lb2) < 0)	copy(ib1,lb1,1);

	while(1) {

		switch(compare(lb1,lb2)) {

			case 0:
				wr(lb1,3);
				if(rd(ib1,lb1) < 0) {
					if(rd(ib2,lb2) < 0)	exit(0);
					copy(ib2,lb2,2);
				}
				if(rd(ib2,lb2) < 0)	copy(ib1,lb1,1);
				continue;

			case 1:
				wr(lb1,1);
				if(rd(ib1,lb1) < 0)	copy(ib2,lb2,2);
				continue;

			case 2:
				wr(lb2,2);
				if(rd(ib2,lb2) < 0)	copy(ib1,lb1,1);
				continue;
		}
	}
}

rd(file,buf)
FILE *file;
char *buf;
{

	register int i, c;
	i = 0;
	while((c = getc(file)) != EOF) {
		*buf = c;
		if(c == '\n' || i > LB-2) {
			*buf = '\0';
			return(0);
		}
		i++;
		buf++;
	}
	return(-1);
}

wr(str,n)
	char  *str;
{
```

```
	switch(n) {

		case 1:
			if(one)	return;
			break;

		case 2:
			if(two)	return;
			break;

		case 3:
			if(three)	return;
	}
	printf("%s%s\n",ldr[n-1],str);
}

copy(ibuf,lbuf,n)
FILE *ibuf;
char *lbuf;
{
	do {
		wr(lbuf,n);
	} while(rd(ibuf,lbuf) >= 0);

	exit(0);
}

compare(a,b)
	char  *a,*b;
{
	register char *ra,*rb;

	ra = --a;
	rb = --b;
	while(*++ra == *++rb)
		if(*ra == '\0')	return(0);
	if(*ra < *rb)	return(1);
	return(2);
}
FILE *openfil(s)
char *s;
{
	FILE *b;
	if(s[0]=='-' && s[1]==0)
		b = stdin;
	else if((b=fopen(s,"r")) == NULL) {
		perror(s);
		exit(1);
	}
	return(b);
}
```

Versões Do Programa COMM

```
=====diff Comm.c and Comm-1.c================
23c23
<           if(*argv[1] == '-' && argv[1][1] != 0) {
---
>           if(*argv[1] != '-' && argv[1][1] != 0) {
=====diff Comm.c and Comm-2.c================
23c23
<           if(*argv[1] == '-' && argv[1][1] != 0) {
---
>           if(*argv[1] == '-') {
=====diff Comm.c and Comm-3.c================
29c29
<                                   one = 1;
---
>                                   one = 2;
=====diff Comm.c and Comm-4.c================
31c31
<                                   ldr[2][l--] = '\0';
---
>                                   ldr[2][l++] = '\0';
=====diff Comm.c and Comm-5.c================
40,42d39
<                       case '3':
<                             three = 1;
<                             break;
=====diff Comm.c and Comm-6.c================
63c63
<           if(rd(ib2,lb2) < 0)     exit(0);
---
>           if(rd(ib2,lb2) <= 0)    exit(0);
=====diff Comm.c and Comm-7.c================
66c66
<     if(rd(ib2,lb2) < 0)     copy(ib1,lb1,1);
---
>     if(rd(ib2,lb2) < 0)     copy(ib1,ib2,1);
=====diff Comm.c and Comm-8.c================
87c87
<                       wr(lb2,2);
---
>                       wr(lb1,2);
=====diff Comm.c and Comm-9.c================
103c103
<           if(c == '\n' || i > LB-2) {
---
>           if(c == '\n' && i > LB-2) {
=====diff Comm.c and Comm-10.c================
103c103
<           if(c == '\n' || i > LB-2) {
---
>           if(c == '\n' || i >= LB-2) {
=====diff Comm.c and Comm-11.c================
108d107
<           buf++;
=====diff Comm.c and Comm-12.c================
121d120
<                  break;
=====diff Comm.c and Comm-13.c================
```

```
127,128d126
<          case 3:
<                if(three)   return;
=====diff Comm.c and Comm-14.c================
139c139
<     } while(rd(ibuf,lbuf) >= 0);
---
>     } while(rd(ibuf,lbuf) > 0);
=====diff Comm.c and Comm-15.c================
151c151
<     while(*++ra == *++rb)
---
>     while(*++ra == *rb++)
=====diff Comm.c and Comm-16.c================
160c160
<     if(s[0]=='-' && s[1]==0)
---
>     if(s[0]=='-' || s[1]==0)
=====diff Comm.c and Comm-17.c================
160c160
<     if(s[0]=='-' && s[1]==0)
---
>     if(s[1]==0)
=====diff Comm.c and Comm-18.c================
153c153
<     if(*ra < *rb)     return(1);
---
>     if(*rb < *ra)     return(1);
=====diff Comm.c and Comm-19.c================
76c76
<                              copy(ib2,lb2,2);
---
>                              copy(ib1,lb1,2);
```

Programa Look

```
/*
 * Copyright (c) 1987 Regents of the University of California.
 * All rights reserved.  The Berkeley software License Agreement
 * specifies the terms and conditions for redistribution.
 */

#ifndef lint
char copyright[] =
"@(#) Copyright (c) 1987 Regents of the University of California.\n\
 All rights reserved.\n";
#endif not lint

#ifndef lint
static char sccsid[] = "@(#)look.c  4.5 (Berkeley) 10/6/87";
#endif not lint

#include <sys/types.h>
#include <sys/file.h>
#include <sys/stat.h>
#include <stdio.h>
#include <ctype.h>

#define     EOS         '\0'
```

```
#define     MAXLINELEN  250
#define     YES         1

static int  fold, dict, len;

main(argc, argv)
      int   argc;
      char  **argv;
{
      extern char *optarg;
      extern int  optind;
      static char *filename = "/usr/dict/words";
      register off_t    bot, mid, top;
      register int      c;
      struct stat sb;
      char  entry[MAXLINELEN], copy[MAXLINELEN];

      while ((c = getopt(argc, argv, "df")) != EOF)
            switch((char)c) {
            case 'd':
                  dict = YES;
                  break;
            case 'f':
                  fold = YES;
                  break;
            case '?':
            default:
                  usage();
            }
      argv += optind;
      argc -= optind;

      switch(argc) {
      case 1:     /* if default file, set to dictionary order and folding */
            dict = fold = YES;
            break;
      case 2:
            filename = argv[1];
            break;
      default:
            usage();
      }

      if (!freopen(filename, "r", stdin)) {
            fprintf(stderr,"look: can't read %s.\n", filename);
            exit(2);
      }
      if (fstat(fileno(stdin), &sb)) {
            perror("look: fstat");
            exit(2);
      }

      len = strlen(*argv);
      canon(*argv, *argv);
      len = strlen(*argv);          /* may have changed */
      if (len > MAXLINELEN - 1) {
            fputs("look: search string is too long.\n", stderr);
            exit(2);
      }
```

```
	for (bot = 0, top = sb.st_size;;) {
		mid = (top + bot) / 2;
		(void)fseek(stdin, mid, L_SET);

		for (++mid; (c = getchar()) != EOF && c != '\n'; ++mid);
		if (!getline(entry))
			break;
		canon(entry, copy);
		if (strncmp(*argv, copy, len) <= 0) {
			if (top <= mid)
				break;
			top = mid;
		}
		else
			bot = mid;
	}
	(void)fseek(stdin, bot, L_SET);
	while (ftell(stdin) < top) {
		register int val;

		if (!getline(entry))
			exit(0);
		canon(entry, copy);
		if (!(val = strncmp(*argv, copy, len))) {
			puts(entry);
			break;
		}
		if (val < 0)
			exit(0);
	}
	while (getline(entry)) {
		canon(entry, copy);
		if (strncmp(*argv, copy, len))
			break;
		puts(entry);
	}
	exit(0);
}

/*
 * getline --
 *	get a line
 */
static
getline(buf)
	register char	*buf;
{
	register int	c;

	for (;;) {
		if ((c = getchar()) == EOF)
			return(0);
		if (c == '\n')
			break;
		*buf++ = c;
	}
	*buf = EOS;
	return(1);
}
```

```
/*
 * canon --
 *    create canonical version of word
 */
static
canon(src, copy)
      register char     *src, *copy;
{
      register int      cnt;
      register char     c;

      for (cnt = len + 1; (c = *src++) && cnt; --cnt)
            if (!dict || isalnum(c))
                  *copy++ = fold && isupper(c) ? tolower(c) : c;
      *copy = EOS;
}

/*
 * usage --
 *    print a usage message and die
 */
static
usage()
{
      fputs("usage: look [-df] string [file]\n", stderr);
      exit(1);
}
```

Versões Do Programa LOOK

```
=====diff Look.c and Look-1.c================
45d44
<                 break;
=====diff Look.c and Look-2.c================
61c61
<           filename = argv[1];
---
>           filename = argv[0];
=====diff Look.c and Look-3.c================
79c79
<     if (len > MAXLINELEN - 1) {
---
>     if (len > MAXLINELEN + 1) {
=====diff Look.c and Look-4.c================
85c85
<           mid = (top + bot) / 2;
---
>           mid = (top + bot) / 2 + 1;
=====diff Look.c and Look-5.c================
118c118
<           puts(entry);
---
>           puts(copy);
=====diff Look.c and Look-6.c================
158c158
<     *copy = EOS;
---
```

```
>     *src = EOS;
=====diff Look.c and Look-7.c================
95c95
<                 top = mid;
---
>                 bot = mid;
=====diff Look.c and Look-8.c================
97d96
<           else
=====diff Look.c and Look-9.c================
101c101
<     while (ftell(stdin) < top) {
---
>     while (ftell(stdin) <= top) {
=====diff Look.c and Look-10.c================
111c111
<           if (val < 0)
---
>           if (val <= 0)
=====diff Look.c and Look-11.c================
114c114
<     while (getline(entry)) {
---
>     while (getline(copy)) {
=====diff Look.c and Look-12.c================
155c155
<     for (cnt = len + 1; (c = *src++) && cnt; --cnt)
---
>     for (cnt = len; (c = *src++) && cnt; --cnt)
=====diff Look.c and Look-13.c================
138c138
<           *buf++ = c;
---
>           *++buf = c;
=====diff Look.c and Look-14.c================
155c155
<     for (cnt = len + 1; (c = *src++) && cnt; --cnt)
---
>     for (cnt = len + 1; (c = *copy++) && cnt; --cnt)
=====diff Look.c and Look-15.c================
27c27
< static int      fold, dict, len;
---
> int fold, dict, len;
=====diff Look.c and Look-16.c================
156c156
<           if (!dict || isalnum(c))
---
>           if (!dict && isalnum(c))
=====diff Look.c and Look-17.c================
157c157
<                 *copy++ = fold && isupper(c) ? tolower(c) : c;
---
>                 *copy++ = fold && (isupper(c) ? tolower(c) : c);
=====diff Look.c and Look-18.c================
47c47
<                 fold = YES;
---
>                 len = YES;
=====diff Look.c and Look-19.c================
```

```
106c106
<           canon(entry, copy);
---
>           canon(copy, entry);
=====diff Look.c and Look-20.c================
58c58
<           dict = fold = YES;
---
>           dict = YES;
```

Programa Uniq

```
static char *sccsid = "@(#)uniq.c   4.1 (Berkeley) 10/1/80";
/*
 * Deal with duplicated lines in a file
 */
#include <stdio.h>
#include <ctype.h>
int   fields;
int   letters;
int   linec;
char  mode;
int   uniq;
char  *skip();

main(argc, argv)
int argc;
char *argv[];
{
      static char b1[1000], b2[1000];

      while(argc > 1) {
            if(*argv[1] == '-') {
                  if (isdigit(argv[1][1]))
                        fields = atoi(&argv[1][1]);
                  else mode = argv[1][1];
                  argc--;
                  argv++;
                  continue;
            }
            if(*argv[1] == '+') {
                  letters = atoi(&argv[1][1]);
                  argc--;
                  argv++;
                  continue;
            }
            if (freopen(argv[1], "r", stdin) == NULL)
                  printe("cannot open %s\n", argv[1]);
            break;
      }
      if(argc > 2 && freopen(argv[2], "w", stdout) == NULL)
            printe("cannot create %s\n", argv[2]);

      if(gline(b1))
            exit(0);
      for(;;) {
            linec++;
            if(gline(b2)) {
                  pline(b1);
```

```
                exit(0);
        }
        if(!equal(b1, b2)) {
                pline(b1);
                linec = 0;
                do {
                        linec++;
                        if(gline(b1)) {
                                pline(b2);
                                exit(0);
                        }
                } while(equal(b1, b2));
                pline(b2);
                linec = 0;
        }
    }
}

gline(buf)
register char buf[];
{
    register c;

    while((c = getchar()) != '\n') {
        if(c == EOF)
            return(1);
        *buf++ = c;
    }
    *buf = 0;
    return(0);
}

pline(buf)
register char buf[];
{

    switch(mode) {

    case 'u':
        if(uniq) {
            uniq = 0;
            return;
        }
        break;

    case 'd':
        if(uniq) break;
        return;

    case 'c':
        printf("%4d ", linec);
    }
    uniq = 0;
    fputs(buf, stdout);
    putchar('\n');
}

equal(b1, b2)
register char b1[], b2[];
{
```

```
      register char c;

      b1 = skip(b1);
      b2 = skip(b2);
      while((c = *b1++) != 0)
            if(c != *b2++) return(0);
      if(*b2 != 0)
            return(0);
      uniq++;
      return(1);
}

char *
skip(s)
register char *s;
{
      register nf, nl;

      nf = nl = 0;
      while(nf++ < fields) {
            while(*s == ' ' || *s == '\t')
                  s++;
            while( !(*s == ' ' || *s == '\t' || *s == 0) )
                  s++;
      }
      while(nl++ < letters && *s != 0)
                  s++;
      return(s);
}

printe(p,s)
char *p,*s;
{
      fprintf(stderr, p, s);
      exit(1);
}
```

Versões Do Programa UNIQ

```
=====diff Uniq.c and Uniq-1.c================
24d23
<                 else mode = argv[1][1];
=====diff Uniq.c and Uniq-2.c================
54c54
<                       linec++;
---
>                       linec--;
=====diff Uniq.c and Uniq-3.c================
57d56
<                             exit(0);
=====diff Uniq.c and Uniq-4.c================
74c74
<           *buf++ = c;
---
>           *++buf = c;
=====diff Uniq.c and Uniq-5.c================
```

```
95d94
<           return;
=====diff Uniq.c and Uniq-6.c================
100c100
<     uniq = 0;
---
>     uniq = 1;
=====diff Uniq.c and Uniq-7.c================
112c112
<     while((c = *b1++) != 0)
---
>     while((c = *b1--) != 0)
=====diff Uniq.c and Uniq-8.c================
127c127
<     while(nf++ < fields) {
---
>     while(nf++ <= fields) {
=====diff Uniq.c and Uniq-9.c================
130c130
<           while( !(*s == ' ' || *s == '\t' || *s == 0) )
---
>           while( !(*s == ' ' || *s == '\t' && *s == 0) )
=====diff Uniq.c and Uniq-10.c================
133c133
<     while(nl++ < letters && *s != 0)
---
>     while(nl++ < letters && s != 0)
=====diff Uniq.c and Uniq-11.c================
18c18
<     static char b1[1000], b2[1000];
---
>     char b1[1000], b2[1000];
=====diff Uniq.c and Uniq-12.c================
114c114
<     if(*b2 != 0)
---
>     if(*b1 != 0)
=====diff Uniq.c and Uniq-13.c================
112c112
<     while((c = *b1++) != 0)
---
>     while((c = *++b1) != 0)
=====diff Uniq.c and Uniq-14.c================
130c130
<           while( !(*s == ' ' || *s == '\t' || *s == 0) )
---
>           while( !(*s == ' ' || *s == '\t'))
=====diff Uniq.c and Uniq-15.c================
130c130
<           while( !(*s == ' ' || *s == '\t' || *s == 0) )
---
>           while( !(*s == ' ' || *s == '\t') || *s == 0 )
=====diff Uniq.c and Uniq-16.c================
61c61
<                 linec = 0;
---
>                 linec = 1;
=====diff Uniq.c and Uniq-17.c================
135c135
<     return(s);
```

```
---
>     return(++s);
=====diff Uniq.c and Uniq-18.c================
88c88
<                 uniq = 0;
---
>                 linec = 0;
=====diff Uniq.c and Uniq-19.c================
91d90
<           break;
```

Processo de Ativação dos Erros do Programa SPACE

Para gerar os programas com erros utilizou-se uma versão do SPACE adequada para essa atividade (prepro.c) utilizando-se o seguinte comando de compilação:

gcc prepro.c –DE$F –o $PROG.exe –lm –w

no qual:

prepro.c: programa fonte que ativa os erros

-DE$F: primitiva que defini qual erro deve ser ativado; $F = erro ativado.