JULIANO ROGERIO TOALDO

UTILIZANDO ANÁLISE DE MUTANTES NO TESTE DE PROGRAMAS PROLOG

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.

Orientadora: Prof.ª Dr.ª Silvia Regina Vergilio

CURITIBA

2003

Ministério da Educação
Universidade Federal do Paraná
Mestrado em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno *Juliano Rogério Toaldo*, avaliamos o trabalho intitulado, "*Utilizando Análise de Mutantes no Teste de Programas Prolog*", cuja defesa foi realizada no dia 25 de setembro de 2003, às dez horas, no Auditório do Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela aprovação do candidato.

Curitiba, 25 de setembro de 2003.

Profª. Dra. Silvia Regina Vergilio
DINF/UFPR - Orientadora

Profª. Dra. Sandra Camargo Pinto Ferraz Fabri
DC/UFSCAR - Membro Externo

Prof. Dr. Alexandre Ibrahim Direne
DINF/UFPR – Membro Interno

MINISTÉRIO DA EDUCAÇÃO - UNIVERSIDADE FEDERAL DO PARANÁ - Setor de Ciências Exatas - Coordenação de Pós-Graduação em Informática

AGRADECIMENTOS

Agradecimentos à professora Dra. Silvia, pela dedicação na orientação deste trabalho, a toda a minha família, em especial a minha esposa Patricia, pelo apoio e incentivo nos momentos mais difíceis, a minha mãe Sonia, ao meu irmão Rene e a minha avó Francisca, pela força e pela alegria por cada etapa vencida. Este trabalho é especialmente dedicado à memória de meu pai Rogerio, que partiu no decorrer deste, deixando um legado de bons exemplos, virtudes e a certeza de que sua falta será para sempre sentida.

RESUMO

Diversos critérios e ferramentas de teste têm sido propostos ultimamente com o objetivo de auxiliar a seleção e a avaliação de um conjunto de dados de teste. Dentre esses critérios, destacam-se os critérios baseados em análise de mutantes, que têm se mostrado um dos mais eficazes em revelar defeitos. Entretanto, esses trabalhos tratam do teste de programas escritos em linguagens procedurais e/ou orientadas a objeto. São poucos os trabalhos da literatura que abordam o teste de programas lógicos, tais como os escritos em Prolog, e a maioria deles não tem como objetivo a quantificação da atividade de teste e o estabelecimento de medidas de cobertura. Este trabalho trata da aplicação do critério Análise de Mutantes no teste de programas Prolog. Um conjunto de operadores de mutação para essa linguagem é proposto. O conjunto está baseado nas principais características do Prolog e em erros comuns que os programadores podem cometer nesse paradigma. Uma ferramenta de teste chamada MutProlog é descrita e resultados de experimentos com essa ferramenta mostram a aplicabilidade dos operadores propostos e permitem a comparação com a abordagem estrutural.

ABSTRACT

Several testing criteria and tools have been proposed lately, with the goal of selecting and evaluating test data sets. The mutation analysis is one of the most important and efficacious testing criterion. However, most of the literature works focus procedural and object-oriented programs and little has been said about logic programming, such as Prolog. Some works address the test of Prolog programs, however does not introduce a testing criterion and do not offer coverage testing metrics. This work investigates the aplication of the mutation analysis criterion for testing Prolog programs. A set of mutation operators for this language is proposed, based on the common mistakes maden by the programmers using this paradigm. A tool, named MutProlog, is described and, results from an experiment, using this tool, show the applicability of the proposed operators and allow comparison with structural criteria.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

CAPÍTULO 1

INTRODUÇÃO

1.1 Contexto

O teste de software constitui-se uma das principais atividades para garantia de qualidade de software. Dada a sua importância, muito trabalho tem sido dedicado com o objetivo de propor novos e eficazes critérios de teste e de implementar ferramentas para auxiliar a automação e a redução dos custos dessa atividade.

Um critério guia a atividade de teste, fornecendo diretrizes para a seleção de dados de teste e medidas de cobertura a serem utilizadas para se considerar a atividade de teste encerrada [25]. Os critérios de teste consideram diferentes aspectos para estabelecer os requisitos de teste. Critérios funcionais utilizam a funcionalidade da aplicação. Critérios estruturais consideram aspectos estruturais tais como fluxo de controle ou fluxo de dados. Critérios baseados em defeitos consideram os principais erros cometidos durante o desenvolvimento.

Nota-se, entretanto, que a maioria dos trabalhos e critérios estão voltados para os paradigmas de programação procedural e/ou orientado a objetos, sendo que poucos autores focalizam linguagens de programação lógica, tais como o Prolog, apesar de esse paradigma ser bastante utilizado. Em particular, observa-se a pouca existência de ferramentas de testes para programas lógicos e, principalmente, para aqueles escritos na linguagem Prolog. Dentre esses poucos trabalhos, muitos investigam a geração de dados de teste a partir da especificação [4, 10, 14, 15] ou utilizando o paradigma de programação lógica indutiva [1], outros concentram-se em depuração [22, 23] e detecção de anomalias[2]. Um trabalho que merece destaque é o de Luo et al [18] que propõe a extensão de critérios estruturais, mais particularmente critérios baseados em fluxo de controle no teste de programas Prolog. Luo et al trabalham com um grafo de fluxo de controle que considera características particulares da linguagem Prolog para derivar os requisitos de teste.

O objetivo do nosso trabalho é investigar a utilização de critérios de teste baseados em defeitos no teste de programas Prolog, mais particularmente do critério Análise de Mutantes [7], nesse contexto. A principal motivação para isso é que o critério Análise de Mutantes tem se mostrado um dos mais eficazes, ou seja, capaz de revelar o maior número de defeitos [21, 28, 29].

Para possibilitar a utilização prática desse critério, faz-se necessário o uso de alguma ferramenta de teste. Estas ferramentas permitem conduzir de maneira automatizada e livre de erros a atividade de teste, além de auxiliarem na comparação empírica entre os diversos critérios. Atualmente, essas ferramentas possibilitam o teste de programas escritos em linguagens imperativas, como FORTRAN [6] e C [8]; não permitem a aplicação do critério em outros paradigmas de programação.

1.2 Motivação

Dado o contexto acima destacam-se os seguintes itens que serviram de motivação para o trabalho.

- A atividade de teste é fundamental para a garantia da qualidade de software e critérios de teste podem não somente auxiliar à geração de dados de teste mas também quantificar o teste, oferecendo medidas de cobertura para avaliar um conjunto de teste.

- A existência de poucos trabalhos estendendo os critérios de teste existentes para serem aplicados no paradigma lógico.

- O Prolog é uma das mais conhecidas e importantes linguagens deste paradigma. A linguagem de programação Prolog foi escolhida por apresentar-se como de grande utilização em diversas áreas, com maior ênfase no campo da pesquisa e desenvolvimento, que necessitam de teste e verificação de qualidade da mesma forma, senão mais, do que uma aplicação puramente comercial. Como exemplo, podemos citar as suas aplicações relacionadas à Inteligência Artificial e utilização de funções

heurísticas, devido à naturalidade com que a linguagem trabalha com árvores de busca e *backtracking*.

- O critério análise de mutantes tem se mostrado um dos mais eficazes e a utilização desse critério no teste de programas Prolog pode aumentar a eficácia do teste. Não existem trabalhos nem ferramentas para permitir isso.

- Para utilização prática do critério análise de mutantes é necessária a definição de operadores de mutação para essa linguagem e a implementação de uma ferramenta.

1.3 Objetivos

Este trabalho introduz um conjunto de operadores de mutação para a linguagem Prolog e descreve uma ferramenta de teste chamada MutProlog que implementa os operadores propostos, permitindo assim que o critério Análise de Mutantes seja também aplicado nesse contexto, com o objetivo principal de aumentar a eficácia da atividade de teste. Além disso, são apresentados resultados de experimentos que analisam não apenas a ferramenta, mas também fazem uma comparação com o critério estrutural.

1.4 Organização da dissertação

O Capítulo 2 apresenta os principais aspectos da atividade de teste, uma descrição do critério análise de mutantes e uma breve descrição dos trabalhos relacionados ao teste de programas Prolog. No Capítulo 3 um modelo de defeitos para os programas Prolog é proposto e baseado nesse modelo um conjunto de operadores de mutação é introduzido. No Capítulo 4 descreve-se a ferramenta MutProlog que fornece suporte ao teste de programas Prolog e implementa o conjunto de operadores proposto. No Capítulo 5 são apresentados resultados de um experimento realizado com o MutProlog. Esses resultados validam o conjunto de operadores e permitem comparação com a abordagem estrutural. O Capítulo 6 contém as conclusões e desdobramentos do trabalho. O Apêndice A apresenta o código fonte dos programas em Prolog utilizados nos experimentos, e suas versões incorretas.

CAPÍTULO 2

TESTE DE SOFTWARE

A atividade de testes apresenta-se como a etapa final do processo de construção de software, tendo por objetivo identificar e eliminar erros que comprometam a qualidade do produto final. Dependendo da área de aplicação do produto, os gastos com testes podem alcançar 40% do custo total de desenvolvimento [24].
Pode-se definir que:

- o teste é a execução de um programa com a intenção de encontrar defeitos;
- um bom caso de teste é aquele que possui grande possibilidade de encontrar defeitos;
- um teste bem sucedido é aquele que detecta um defeito que ainda não tinha sido encontrado.

Os testes de software podem ser classificados em 3 diferentes tipos ou níveis que necessariamente devem existir em uma estratégia real de um produto: teste de unidade, teste de integração e teste de sistema. O teste de unidade é executado sobre a menor unidade do projeto de software, o módulo, sendo que o objetivo deste teste é identificar erros de lógica, implementação e inconformidade de resultados gerados em relação à definição ou interface. O teste de integração é realizado para encontrar problemas que podem aparecer na integração dos componentes, sendo que geralmente são encontrados defeitos na interface entre os módulos. Já o teste de sistema é realizado após os testes de unidade e integração e tem por objetivo encontrar erros em relação a funções e características de desempenho que não estejam de acordo com a especificação, além de problemas que podem surgir relacionados à integração do produto com sistemas operacionais, banco de dados, dentre outros fatores externos ao software.

Um dos grandes problemas da atividade de teste é definir quais casos de teste devem ser utilizados de maneira que os defeitos possam ser encontrados, mas o número de testes

necessários não pode tornar impraticável a realização da atividade.

Dentro do escopo deste trabalho, a terminologia usada em relação a erro, falha e defeito está relacionada à seguinte definição [17]:

- Erro: item de informação ou estado de execução inconsistente;
- Defeito: deficiência mecânica ou algorítmica que se efetivada pode levar a uma falha;
- Falha: evento notável em que o sistema viola suas especificações, produzindo uma saída incorreta e/ou não esperada.

Os critérios de teste podem ser aplicados para facilitar a atividade de teste de software. Os critérios têm sido definidos com o objetivo de fornecer uma maneira eficiente de selecionar pontos do domínio de entrada do programa e conseguir revelar defeitos existentes, permitindo que a atividade de teste enquadre-se nas restrições de tempo e custo de um projeto de software. Além disso, eles oferecem medidas de cobertura de um dado conjunto de teste, avaliando assim a atividade de teste e permitindo o seu encerramento.

Atualmente, os critérios podem ser agrupados dentro das três seguintes técnicas de teste: Técnica Funcional, Técnica Estrutural e Técnica Baseada em Erros. Os principais critérios serão descritos a seguir.

2.1 Critérios de teste

2.1.1 Critérios funcionais

A técnica funcional, também conhecida como teste de Caixa Preta ou Black-Box, testa o software como um componente com a estrutura desconhecida, sendo que somente sua interface, entradas e saídas, são conhecidas. O objetivo deste teste é verificar as funcionalidades do componente, sem se preocupar com sua implementação. Essas funcionalidades devem estar de acordo com a especificação do componente e de acordo com os requisitos do usuário.

Os critérios de teste funcional mais conhecidos são: Particionamento em Classes de Equivalência, Análise do Valor Limite e Grafos de Causa e Efeito [24]. Particionamento em

Classes de Equivalência é um critério de teste funcional que divide o domínio de entrada de um programa em classes, a partir das quais os casos de teste podem ser derivados.

O objetivo, através da melhor escolha dos casos de teste com o auxílio das classes, 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 existente.

O critério Análise do Valor Limite é um complemento ao critério Particionamento em Classes de Equivalência, onde procuram-se testar os limites de cada classes de equivalência. Segundo Pressman [24], erros costumam ocorrer com mais frequência nos limites dos domínios de entrada do que no centro desses domínios, tornando o critério Análise do Valor Limite relevante para o teste funcional de um sistema. Utiliza-se o critério funcional Grafo de Causa e Efeito quando é necessário 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.

Em geral, o teste funcional é uma técnica de validação de programas onde os casos de teste são gerados a partir da especificação dos requisitos tornando-se uma técnica sujeita às inconsistências que podem ocorrer na especificação [16]. Além disso, como não existe conhecimento sobre os pontos críticos da implementação, não é possível garantir que estes pontos serão testados.

2.1.2 Critérios estruturais

A técnica estrutural baseia-se no conhecimento da estrutura interna da implementação ou especificação. A maioria dos critérios dessa técnica utiliza uma representação de programa conhecida como grafo de fluxo de controle ou grafo de programa. Um grafo de fluxo de controle é um grafo orientado onde cada vértice representa um bloco indivisível de comandos e cada aresta representa um desvio de um bloco para outro, sendo que 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 sequencialmente. Através do grafo escolhem-se os componentes que devem ser executados, 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, sendo estes os Critérios Baseados em Fluxo de Controle, Baseados em Fluxo de Dados e Baseados na Complexidade.

O critério baseado na complexidade utiliza informações sobre a complexidade do programa para determinar os requisitos de teste. Um critério bastante conhecido desta classe é o critério de McCabe que utiliza a complexidade ciclomática para derivar os requisitos de teste, sendo que é necessário executar um conjunto de caminhos linearmente independente do grafo de programa [24].

Os critérios baseados em fluxos de controle usam apenas características de controle de execução do programa, como comandos ou desvios, para determinar quais estruturas são necessárias. Os critérios mais conhecidos são os critérios todos-arcos, todos-nós e todos-caminhos. Os critérios todos-nós e todos-arcos exigem que durante a execução do programa, exercite-se pelo menos uma vez cada nó e vértice do grafo de fluxo (ou que cada comando do programa seja executado pelo menos uma vez), e cada aresta do grafo (cada desvio de fluxo no programa). Já o critério todos-caminhos que, em geral, é impraticável, requer que todos os caminhos, sequências possíveis de nós do grafo, sejam executados [24]. 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 [12].

Já a classe de critérios baseados em fluxo de dados requer que sejam testadas as interações que envolvem definições de variáveis e referências a estas definições [26]. Exemplos dessa classe de critérios são os critérios de Rapps & Weyuker [25, 26] e os critérios de Maldonado et al [20].

Rapps & Weyuker propuseram o grafo def-uso (Def-Use Graph) que consiste de uma extensão do grafo de programa [25, 26]. Nesse grafo são associadas informações sobre o fluxo de dados do programa, caracterizando associações entre pontos do programa onde

é atribuído um valor a uma variável (definição da variável) e pontos onde esse valor é utilizado (referência ou uso da variável). O uso da variável pode ser computacional (c-uso), quando a variável é usada em uma computação, e predicativo (p-uso), quando usada em uma condição. Com base nessas associações, são determinados os requisitos de teste. Os critérios principais de Rapps & Weyuker são:

- Todas-definições (all-defs): requer que cada definição de variável seja exercitada pelo menos por um c-uso ou um p-uso.

- Todos-usos (all-uses): requer que todas as associações entre uma definição de variável x e c-usos e p-usos da mesma sejam exercitadas pelos casos de teste, através de pelo menos um caminho livre de definição (caminhos onde não ocorre redefinição de x).

- Todos-du-caminhos (all-du-paths): requer que toda associação entre uma definição de variável e c-usos e p-usos dessa variável seja exercitada por todos os caminhos livres de definição e de laço que cubram essa associação.

Os critérios Potenciais-Usos requerem associações independentemente da ocorrência explícita de uma referência a uma determinada variável, ou seja, requer que caminhos livres de definição a partir de uma determinada variável sejam executados, independentemente de ocorrer uso dessa variável nesse caminho [19]. Os critérios básicos que fazem parte dessa família de critérios são:

- Todos-potenciais-usos: basicamente, requerem que pelo menos um caminho livre de definição de uma variável definida em um nó i para todo nó e todo arco possível de ser alcançado a partir de i seja exercitado.

- Todos-potenciais-du-caminhos: requerem que todos os potencias-du-caminhos com relação a todas as variáveis x definidas e todos os nós e arcos possíveis de serem alcançados a partir dessa definição sejam exercitados. Um potencial-du-caminho em relação à variável x é um caminho livre de definição (n_1,...,n_j, n_k) com relação a x do nó n_1 para o nó n_k e para o arco (n_j, n_k), onde o caminho (n_1,...,n_j) é um caminho livre de laço e no nó n_1 ocorre uma definição de x.

Um problema que ocorre com os critérios da técnica estrutural é a impossibilidade de determinar automaticamente se um caminho é ou não executável, ou seja, não existe algoritmo que dado um caminho completo qualquer, forneça o conjunto de valores que causam a execução desse caminho [13]. Dessa forma, uma dificuldade existente é a necessidade de se determinar manualmente quais são os caminhos não executáveis para o programa em teste.

2.1.3 Critérios baseados em erros

A técnica de teste baseada em erros utiliza informações sobre os tipos mais comuns de erros cometidos durante o processo de desenvolvimento de um software. A ênfase desta técnica está nos erros que o programador ou projetista pode cometer durante o processo de desenvolvimento, e nas abordagens que podem ser usadas para detectar a sua ocorrência. A Análise de Mutantes [7] e a Semeadura de Erros [3] são critérios típicos que se concentram em erros.

No critério semeadura de erros, alguns erros são inseridos "artificialmente" no programa. Então, dos erros encontrados durante o teste, verificam-se quais são naturais e quais são artificiais. A razão dos erros artificiais pelos naturais representa, teoricamente, o número de erros naturais ainda existentes no programa. Um problema com esse critério está no fato de que alguns erros naturais sejam omitidos pelos erros semeados [3].

A Análise de Mutantes(ou teste de mutantes) é um critério de teste baseado em erros que, para avaliar o quanto um conjunto de casos de teste T é adequado para o teste de um dado programa P, utiliza 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 [7]. Na próxima seção apresentamos este critério de forma mais detalhada, por ele ser o foco deste trabalho.

2.2 O Critério análise de mutantes

A característica da técnica baseada em erros é utilizar-se dos erros mais comuns que ocorrem durante o processo de desenvolvimento de software para derivar os casos de teste, verificando assim se estes erros estão presentes ou não no código a ser analisado. Dentre as técnicas baseadas em erros, o critério Análise de Mutantes apresenta-se como um dos mais promissores.

O Critério de Mutantes está fundamentado em dois pressupostos estabelecidos por DeMillo [7]:

- Hipótese do programador competente: baseia-se no fato de que os programadores, em geral, não cometem grandes enganos no desenvolvimento, mas produzem programas muito próximos do ideal, que potencialmente possui pequenos erros. Assim, seriam introduzidos defeitos em um programa, gerando alterações sintáticas válidas, com o objetivo de determinar quais seriam as inconformidades.

- Efeito do acoplamento: supõe que "defeitos complexos são decorrentes de defeitos simples", ou seja, um pequeno defeito em um sistema acabaria desencadeando uma sequência de defeitos que iriam gerar um problema mais complexo. Assim, pode-se supor que exista um conjunto de testes que detecte defeitos complexos a partir de defeitos simples.

O teste de mutação pode ser descrito da seguinte forma: inicialmente, para um certo programa P, são gerados vários programas P_1, P_2, ..., P_n, cuja diferença em relação ao programa original são pequenas alterações sintáticas para simular defeitos simples. Cada um desses programas alterados P_i são chamados de mutantes de P. Para que o critério seja satisfeito, é necessário gerar um dado de teste t que faça com que o resultado do mutante seja diferente do resultado gerado pelo programa original. Se isto acontecer, é dito que o mutante P_i está morto. Se não existir dado de teste que possa distinguir um mutante P_i do programa P que está sendo testado, então dizemos que P_i é equivalente a P.

Um dos grandes problemas da Análise de Mutantes é o alto custo computacional necessário para executar a técnica, devido ao grande número de mutantes que devem ser gerados para satisfazer o critério. Além disso, a existência de mutantes equivalentes que necessitam da intervenção humana para serem detectados dificultam a utilização da técnica na prática. Algumas heurísticas vem sendo desenvolvidas com o objetivo de minimizar estes problemas [5].

Um mutante equivalente é aquele que apresenta comportamento igual ao programa original, sendo que neste passo a intervenção humana é necessária.

Um exemplo que podemos citar é:

Programa original:

```
...
if (a>=b)
   <comandos>
...
```

Mutante equivalente:

```
...
if ((a>b)||(a=b))
<comandos>
...
```

Neste caso, os dois programas, tanto o original como o mutante, irão gerar a mesma saída, independentemente da entrada.

Além de possibilitar a geração de um conjunto de dados de teste T, o critério Análise de Mutantes pode ser utilizado para avaliar a adequação de um conjunto de testes.

O escore de mutação é obtido através de uma relação entre o número de mutantes mortos e o número total de mutantes gerados, permitindo a análise da adequação do conjunto T utilizado, tal como apresentado abaixo.

$$S_M(P,T) = \frac{M_d(P,T)}{M(P) - M_e(P)}, onde:$$

- P: programa em teste;
- T: conjunto de dado de teste;
- $S_M(P,T)$: escore de mutação (cobertura);
- $M_d(P,T)$: número de mutantes mortos;
- $M(P)$: número total de mutantes gerados;
- $M_e(P)$: número de mutantes equivalentes.

O escore é dado em termos do número de mutantes mortos e do número de mutantes gerados a partir de P. Devido à existência de mutantes equivalentes, a cobertura do critério fica comprometida, sendo geralmente menor que a unidade, se os mutantes equivalentes não forem marcados.

Em relação a aplicação na prática do critério, utiliza-se o conceito de operadores de mutação. O operador de mutação apresenta-se como uma transformação sintática que é aplicada em alguma estrutura em um programa P, trocando o seu conteúdo com o objetivo de verificar o comportamento de P em relação a esta mudança.

As primeiras ferramentas que automatizaram a aplicação do critério Análise de Mutantes começaram a aparecer a partir de 1979. Estas ferramentas eram direcionadas ao teste de programas desenvolvidos em Fortran e Cobol.

A partir dos anos 80, ferramentas mais avançadas começaram a surgir, como a ferramenta Mothra que é utilizada para testes de programas escritos em Fortran [6].

A ferramenta Proteum(Program Testing Using Mutants) é uma ferramenta de teste para programas escritos na linguagem C, desenvolvida pelo Grupo de Testes do Instituto de Ciências Matemáticas de São Carlos [9].

Todas as ferramentas citadas acima utilizam operadores de mutação para linguagens procedurais, não permitindo o teste de programas Prolog ou outras linguagens similares.

2.2.1 Ferramenta Proteum

A ferramenta Proteum é uma ferramenta de apoio ao critério Análise de Mutantes para programas escritos em linguagem C que será utilizada como base para o nosso trabalho. Ela implementa os seguintes procedimentos:

- definição de casos de teste;
- execução do programa em teste sem alteração;
- alteração da porcentagem de aplicação dos operadores de mutação, que serão utilizados para gerar os mutantes;
- execução dos mutantes com os casos de teste definidos;
- análise dos mutantes que sobraram(equivalentes ou que restaram devido a testes que não os mataram) e cálculo do escore de mutação.

Na Proteum utiliza-se a idéia de sessão de teste para relacionar todas as atividades que dizem respeito ao teste, como a elaboração do teste em etapas que podem ter armazenados os seus estados intermediários para que a atividade seja retomada mais tarde. Desta forma, o usuário pode iniciar o teste de um programa, não criar ou executar todos os mutantes, parar a atividade, e depois retomar novamente a sessão de teste a partir do ponto em que foi deixada, sem a necessidade de reiniciá-la.

Os operadores de mutação definidos e disponíveis na ferramenta são divididos em 4 classes: mutação sobre comandos (statement mutations), mutação sobre operadores (operator mutations), mutação sobre variáveis (variable mutations) e mutação sobre constante (constant mutations). Esses operadores podem ser aplicados de acordo com as classes de erros que se deseja enfatizar ou analisar em particular no programa em teste. Isto acaba possibilitando a geração de mutantes em etapas, onde essa atividade pode ser dividida entre vários testadores trabalhando independentemente.

A execução dos mutantes é feita independente da intervenção do testador onde, à medida que os mutantes são executados, é comparada a saída obtida dos mutantes com a saída do programa teste. É baseada nessa saída que é realizada a análise da equivalência.

Normalmente um mutante é executado pelos casos de teste até que um deles o mate; caso isso não ocorra, o mutante é dito vivo e posteriormente pode ser analisado para determinar sua equivalência ou não.

Uma característica adicional da Proteum é a possibilidade de executar um mutante com todos os casos de teste disponíveis, mesmo que algum deles já o tenha matado, permitindo determinar quais casos de teste são mais eficientes.

Após executados os mutantes, o usuário pode verificar o escore de mutação obtido e decidir se o teste deve continuar ou não, adicionando mais casos de teste que potencialmente podem apresentar melhor escore. Para melhorar o escore de mutação o testador irá verificar os mutantes vivos e decidir pela sua equivalência ou não. Caso não sejam equivalentes, novos casos de teste podem ser adicionados executando-se os mutantes com esses novos casos de teste, onde somente os mutantes vivos serão executados.

Pode-se também verificar o andamento da atividade através de um relatório estatístico sobre os casos de teste, o qual pode apresentar informações tais como: total de mutantes gerados, executados, ativos, vivos e equivalentes; escore de mutação; operadores de mutação e as porcentagens selecionadas; número de mutantes mortos para cada caso de testes dentre outras.

A ferramenta também permite a construção de "scripts" para executar os testes, economizando tempo na atividade de teste com a redução do número de interações com a ferramenta.

2.2.2 Ferramenta Mothra

Outra ferramenta que utiliza mutação para fazer testes é a Mothra, que testa programas escritos em Fortran-77 [6]. Os programas são automaticamente traduzidos para o "front-end" do compilador dentro de um formato que consiste de uma tabela de símbolos flexíveis e tuplas do Código Intermediário do Mothra (MIC). Casos de teste são armazenados e o programa é executado através da interpretação do formato interno. O formato interno também é utilizado para gerar mutantes, que são armazenados como registros que descrevem as mudanças no MIC necessárias para criar mutantes. A ferramenta Mothra

analisa cada mutante vivo contra cada caso de teste e reporta os resultados.

2.3 Comparação entre critérios

Estudos teóricos e empíricos permitem a comparação dos diversos critérios de teste existentes e têm sido conduzidos com o objetivo de se encontrar formas econômicas e produtivas para se produzir testes. Segundo Wong [29] custo, eficácia e dificuldade de satisfação (*strength*) são os fatores básicos para comparar a adequação dos critérios de teste:

- Custo: refere-se ao esforço necessário para utilizar o critério e é geralmente medido pelo número de casos de teste necessários para satisfazer o critério dado;
- Eficácia: refere-se à capacidade de um critério em revelar um número maior de defeitos em relação a outro;
- Dificuldade de satisfação: refere-se à probabilidade de satisfazer um critério tendo satisfeito outro.

Uma visão geral dos estudos realizados pode ser encontrada em [27]. Esses trabalhos apontam que o Critério Análise de Mutantes, um critério baseado em erros, é o mais eficaz em revelar defeitos, porém o mais custoso.

2.4 Teste de programas Prolog

Percebe-se que nas últimas décadas muito trabalho tem sido dedicado à atividade de teste bem como ao estabelecimento de critérios e à implementação de ferramentas de teste. Entretanto, a maioria deles focalizam o teste de programas escritos em linguagens imperativas e/ou orientadas a objeto. Dentre os trabalhos que visam a linguagem Prolog, a maioria não tem como objetivo o estabelecimento de critérios de teste e não oferecem medidas para quantificar essa atividade. Os trabalhos descritos em [4, 10, 14, 15] investigam a geração de dados de teste a partir da especificação. Outros concentram-se em depuração [22, 23] e detecção de anomalias[2]. Um trabalho mais recente de Bergadano e Gunetti [1] também visa à geração de dados de teste utilizando o paradigma de Programação Lógica

Indutiva (ILP (Inductive Logic Programming)), sem fornecer entretanto nenhuma medida de cobertura. Outro trabalho que merece destaque e que propõe, similarmente ao nosso, a extensão de critérios de teste para programas Prolog é o de Luo et al [18]. A seguir esses dois últimos trabalhos são sumarizados.

2.4.1 Programação Lógica Indutiva - ILP(Inductive Logic Programming)

Dado um programa P e um conjunto de programas alternativos A, gera-se uma sequência de casos de teste que são adequados, no sentido que distinguem o programa dado dos alternativos. O método está relacionado a técnicas baseadas em erro para geração de casos de teste, mas programas em A não precisam ser simples mutações de P.

A técnica para gerar um conjunto de testes adequado é baseado no aprendizado indutivo de programas (ou *Inductive Logic Programming*) [7] a partir de conjuntos finitos de exemplos de entrada e saída. Dado um conjunto parcial de teste, induz-se um programa P' que pertence a A que é consistente a P em relação aos valores de entrada. A partir deste ponto é feita uma busca por um valor de entrada que possa distinguir P de P' e o processo é repetido até que nenhum programa, exceto P, possa ser induzido a partir dos exemplos gerados. É mostrado que o conjunto de teste obtido é adequado em relação aos programas alternativos que pertencem a A.

O método é possível por um procedimento de indução de programa que tem evoluído a partir de pesquisas recentes em aprendizado de máquina e programação lógica indutiva. Uma alternativa é gerar o programa utilizando-se Programação Genética. Procedimentos de teste utilizando Programação Genética para o teste de programas são apresentados em [11].

2.4.2 Teste de programas em Prolog baseado em fluxo de controle

Luo et al [18] propõem um modelo de defeitos para programas em Prolog, que podem ser problemas textuais ou baseados na estrutura sintática dos programas. Este modelo de defeitos serve como um guia para desenvolver critérios de seleção de teste e está baseado nas seguintes características dos programas Prolog: as estruturas de dados do Prolog são listas recursivas, as unificações dos sub-goals em Prolog podem proceder em duas direções, alterando o fluxo de controle e de dados, que é bastante diferente das linguagens tradicionais.

Guiado pelo modelo baseado em defeitos, são fornecidos critérios de seleção em termos de cobertura de fluxo de controle. Embora o teste de programas baseado em fluxo de controle seja comum, esta técnica em Prolog é mais complicada, pela natureza declarativa da linguagem. Para isso, são apresentados dois grafos chamados *P-flowgraph* e *reduced global P-flowgraph* para representar a estrutura intrínsica de fluxo de controle em programas Prolog.

O *P-flowgraph* representa o fluxo de controle de um certo predicado no nível mais alto. O outro grafo, o *reduced global P-flowgraph*, representa a parte do fluxo de controle global em um conjunto de predicados que pode ser chamado por um dado predicado.

Os critérios são, no caso do *P-flowgraph*, a cobertura dos ramos (para cada predicado, gerar um conjunto de dados de teste que cubra todos os seus ramos) e a cobertura dos pares ramo-a-ramo(para cada predicado, gerar um conjunto de dados de teste que cubra todos os pares de ramos do grafo). Para o caso do *reduced global P-flowgraph*, o critério proposto é a cobertura de todos os ramos do grafo.

A Figura 2.1 apresenta o programa *num_ap* e a Figura 2.2 o grafo a ele associado, neste caso estamos trabalhando com o *P-flowgraph*. O caso de teste com a entrada *[1,1,2,2,3,3,4]* exercita todos os ramos do grafo.

Para validar suas idéias, os autores implementam uma ferramenta chamada TGT (Test Data Generation Tool) que instrumenta o código Prolog e monitora a sua execução.

```
num_ap(Lista,Res):-
        num_ap2(Lista,[],Res).

num_ap2([],Res,Res).
num_ap2([X|Resto],L,Res):-
        not(member([X,_],L)),!,
        num_occ(X,[X|Resto],N),
        num_ap2(Resto,[[X,N]|L],Res).
num_ap2([_|Resto],L,Res):-
        num_ap2(Resto,L,Res).

num_occ(_,[],0).
num_occ(X,[X|Resto],Num):-
        num_occ(X,Resto,N),
        Num is N+1.
num_occ(X,[Y|Resto],Num):-
        X \== Y,
        num_occ(X,Resto,Num).
```

Figura 2.1: Código fonte programa *num_ap*

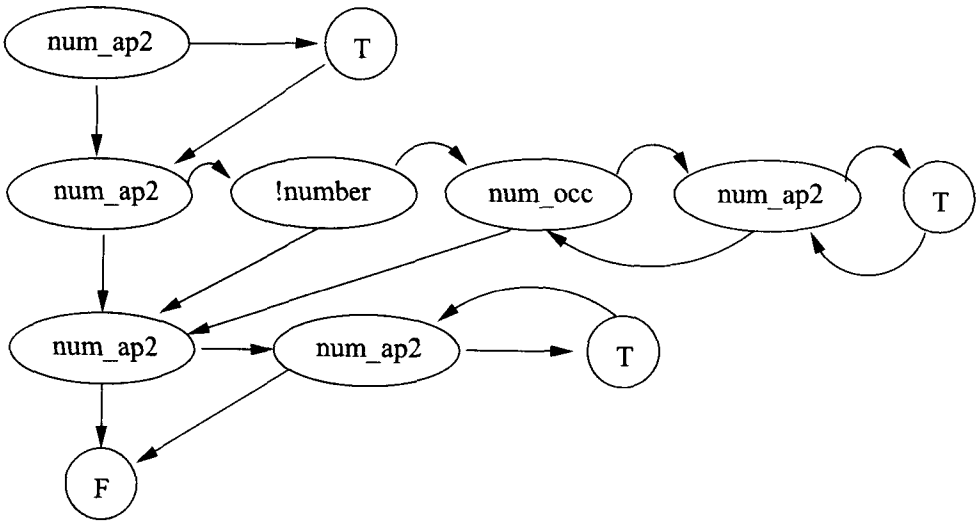

Figura 2.2: *P-flowgraph* do programa *num_ap*

2.5 Considerações finais

Neste capítulo foi apresentada uma visão geral sobre o teste de programas, principalmente terminologia, técnicas e critérios existentes. Também foram apresentados os principais trabalhos que focalizam o teste de programas Prolog e que têm por objetivo propor a extensão dos critérios existentes para serem aplicados nesse paradigma.

Os critérios funcionais, estruturais e baseados em defeitos são considerados complementares, pois podem revelar diferente tipos de defeitos [24]. Muitos trabalhos comparando empiricamente critérios de teste mostram que o critério Análise de Mutantes é um dos mais eficazes em revelar defeitos [21, 28, 29], por isso é bastante importante que esse critério seja também utilizado no teste de programas Prolog. Entretanto, não existem relatos de trabalhos com esse objetivo. Para aplicação do critério Análise de Mutantes no teste de programas Prolog, o primeiro passo é definir um conjunto de operadores de mutação para essa linguagem e implementar uma ferramenta de teste que auxilie a geração, execução e avaliação dos mutantes gerados a partir dos operadores propostos. Os próximos capítulos tratam especificamente desses passos.

CAPÍTULO 3

DEFINIÇÃO DOS OPERADORES DE MUTAÇÃO

Neste capítulo é introduzido um conjunto de operadores de mutação para a linguagem Prolog. Para determinar esse conjunto, além de observações decorrentes de nossa experiência como programadores Prolog, foram consideradas as principais características de um programa Prolog mencionadas na Seção 2.4.2 e apontadas por Luo et al [18]. Além da definição dos operadores, são apresentados exemplos de sua aplicação e uma análise do modo como estes operadores podem auxiliar no processo de busca de defeitos.

3.1 Descrição dos operadores

As seguintes características dos programas em Prolog foram levadas em consideração na determinação dos operadores:

- as estruturas de dados são listas recursivas. A recursão é muitíssimo utilizada em Prolog.
- o fluxo de execução ocorre em dois sentidos. O fluxo continua para frente após o sucesso durante uma unificação ou pode retroceder ("backtracking") após o insucesso.
- variáveis não possuem tipo e podem ser instanciadas com qualquer valor. Existe também a variável anônima.
- não se utilizam rotinas, como em outras linguagens, mas sim um conjunto de cláusulas (regras) a serem utilizadas, sendo difícil caracterizar uma unidade.

Essas características, por sua vez, estão relacionadas com as principais dificuldades para se escrever um programa Prolog e aos principais enganos cometidos pelos programadores que, muitas vezes, não possuem o controle do programa como acontece em outros paradigmas de programação.

Considerando o exposto acima, um modelo de defeitos que se traduz em uma classificação dos principais defeitos encontrados nos programas Prolog é introduzido, estabelecendo, por conseqüência, um operador de mutação para cada classe. Os operadores não têm como objetivo representar erros sintáticos que podem ser detectados no processo de compilação. Os operadores foram agrupados em quatro diferentes grupos: mutação em cláusulas, mutação em operadores, mutação em variáveis, mutação em constantes.

Mutação em Cláusulas

Esses operadores têm como objetivo realizar mutações em cláusulas do programa em teste. Cada regra Prolog, finalizada com um ".", é considerada uma cláusula C. Os operadores desse grupo podem ser visualizados na Tabela 3.1 e incluem: remoção de predicados, troca de ordem de predicados, troca disjunção por conjunção e vice-versa e mutações com "cut" (!). Essa tabela inclui a descrição dos operadores e um exemplo, mostrando na coluna Saída a transformação causada pelo operador. Para evitar a geração de mutantes que seriam facilmente mortos, ou seja, para manter a hipótese do programador compentente, sugerimos que o operador "Swap Predicate" somente realize troca em predicados adjacentes em C, combinando-os dois a dois e o operador "Permut Cut" deve realizar todas as combinações possíveis em C, pois a incorreta colocação do cut em uma cláusula C acontece freqüentemente.

Mutação em Operadores

Ao contrário de linguagens tradicionais, poucos são os operadores da linguagem Prolog, incluímos os aritméticos e relacionais. Os operadores desse grupo estão na Tabela 3.2. A idéia é realizar a troca entre operadores do mesmo tipo disponíveis na linguagem Prolog.

Mutação em Variáveis

Os operadores de mutação em variáveis consideram dois tipos de variáveis, as "normais" e as variáveis "anônimas" representadas pelo "_" em Prolog. Exemplos podem ser visualizados na Tabela 3.3. Uma variável em C será trocada por uma outra variável,

Tabela 3.1: Mutação em Cláusulas

Operador	**Descrição**	**Entrada**	**Saída**
Predicate Deletion	remove um predicado P em uma cláusula C.	writel([]). writel([H|T]) :- write(H),nl, writel(T).	writel([]). writel([H|T]) :- nl, writel(T).
Swap Predicates	troca a ordem de predicados adjacentes P_1 e P_2 em uma cláusula C.	likes(ana,X) :- toy(X), plays(ana,X).	likes(ana,X) :- plays(ana,X), toy(X).
Conjunction by Disjuction Replacement	troca a conjunção entre predicados P_1 e P_2 por uma disjunção em uma cláusula C.	subset(S,[H|T]) :- subset(R,T), (S=R , S=[H|R]). subset([],[]).	subset(S,[H|T]) :- subset(R,T), (S=R ; S=[H|R]). subset([],[]).
Disjunction by Conjunction Replacement	troca a disjunção entre predicados P_1 e P_2 em uma cláusula C.	least_num(X,[H|T]) :- least_num(Y,T), (H=<Y, X=H ; H>Y, X=Y).	least_num(X,[H|T]) :- least_num(Y,T), (H=<Y, X=H , H>Y, X=Y).
Insert Cut	insere um "cut" (!) entre os predicados P_1 e P_2 em uma cláusula C.	bubble_sort(L,L1) :- swap(L,L2,0), bubble_sort(L2,L1). bubble_sort(L,L).	bubble_sort(L,L1) :- swap(L,L2,0),!, bubble_sort(L2,L1). bubble_sort(L,L).
Remove Cut	remove uma ocorrência do "cut" (!) em uma cláusula C.	not(G) :- G,!,fail. not(G).	not(G) :- G,fail. not(G).
Permute Cut	permuta uma ocorrência do "cut" (!) em uma cláusula C.	insert(X,[H|T],[H|T1] :- !,X>H, insert(X,T,T1).	insert(X,[H|T],[H|T1] :- X>H,!, insert(X,T,T1).

Tabela 3.2: Mutação em Operadores

Operador	**Descrição**	**Entrada**	**Saída**
Arithmetic Operator Mutation	troca um operador aritmético por outro operador aritmético	length([],0). length([_|T],N) :- length(T,M),N is M+1.	length([],0). length([_|T],N) :- length(T,M),N is M*1.
Relational Operator	troca um operador relacional por outro operador relacional	fault(X) :- non(respond(X,Y)),X\ ==Y.	fault(X) :- non(respond(X,Y)),X>Y.

Tabela 3.3: Mutação em Variáveis

Operador	Descrição	Entrada	Saída
Variable by Variable	troca uma variável em uma cláusula C por uma outra em C	member(X,[T|_]). member(X,[_|T]) :- member(X,T).	member(X,[X|_]). member(X,[_|T]) :- member(X,T).
Variable by Anonymous Variable	troca uma variável em uma cláusula C por uma variável anônima de C	length([],0). length([H|T],N) :- length(T,M),N is M+1.	length([],0). length([_|T],N) :- length(T,M),N is M+1.
Anonymous Variable by Variable	troca uma variável anônima em uma cláusula C por uma variável em C	member(X,[X|_]). member(X,[_|T]) :- member(X,T).	member(X,[X|X]). member(X,[_|T]) :- member(X,T).

Tabela 3.4: Mutação em Constantes

Operador	Descrição	Entrada	Saída
Constant by Constant	troca uma constante em uma cláusula C por uma outra constante	mult([],[]). mult([X|P],[N1,N2|Y]):- mult(P,Y), N1 is X+1, N2 is X+2.	mult([],[]). mult([X|P],[N1,N2|Y]):- mult(P,Y), N1 is X+1, N2 is X+1.
Constant by Anonymous Variable	troca uma constante em uma cláusula C por uma variável anônima	likes(ana,jose). likes(jose,ana).	likes(ana,jose). likes(jose,_).
Anonymous Variable by Constant	troca uma variável anônima em uma cláusula C por uma constante	likes(ana,apple). likes(paulo,_).	likes(ana,apple). likes(paulo,paulo).
Constant by Variable	troca uma constante em um predicado P por uma variável	nth(N,X,[X|T]). nth(N,X,[Y|T]) :- nth(M,X,T),N is M+1.	nth(N,X,[X|T]). nth(N,X,[Y|T]) :- nth(M,X,T),N is M+N.
Variable by Constant	troca uma variável em uma cláusula C por uma constante	nth(N,X,[X|T]). nth(N,X,[Y|T]) :- nth(M,X,T),N is M+1.	nth(N,X,[X|T]). nth(1,X,[Y|T]) :- nth(M,X,T),N is M+1.

anônima ou não, disponível na mesma cláusula C.

Mutação em Constantes

Os operadores de mutação em constantes realizam substituições de constantes em C por outras constantes ou variáveis (anônimas ou não) presentes na mesma cláusula C. Exemplos podem ser visualizados na Tabela 3.4.

3.2 Exemplos de Uso dos Operadores

Para exemplificar os operadores que compõem as mutações em cláusulas, vamos utilizar os programas *smallest, lenght, member, subset, mult e least_number*, cujos códigos e funcionalidades são os seguintes:

```
smallest([A],A) :- !.
smallest([A|B],N) :- smallest(B,N), N<A, !.
smallest([A| _ ], A).
```

Este programa possui como entrada uma lista de números, e a saída é o elemento que possui o menor valor.

Outro programa que vamos utilizar como exemplo é o *length*, que calcula o tamanho de uma lista:

```
length([], 0).
length([A|B],N) :- length(B,M), N is M+1 .
```

O programa *member* também é utilizado como exemplo, sendo que o programa verifica se o elemento pertence a lista:

```
member(X,[X|_]).
member(X,[_|T]) :- member(X,T).
```

Outro programa utilizado como exemplo, o *subset*, verifica se uma lista é um subconjunto de outra:

```
subset(S,[H|T]):-subset(R,T), (S=R, S=[H|R]).
subset([],[]).
```

Também temos como exemplo o programa *mult*, que simplesmente retorna uma lista com os valores da lista de entrada multiplicada por 2 e por 3:

```
mult([],[]).
mult([X|P],[N1,N2|Y]):-mult(P,Y), N1 is X+1, N2 is X+2.
```

Por último, utilizamos o programa *least_num*, que encontra o menor elemento da lista:

```
least_num(X,[H|T]) :- least_num(Y,T), (H=<Y, X=H ; H>Y, X=Y).
```

Além desses programas, é utilizada uma cláusula que apenas inicializa uma variável:

```
inicializa(0,0,_).
```

3.2.1 Mutação em cláusulas (Clause Mutation)

3.2.1.1 Retirada de predicados(Predicate Deletion)

Remove um predicado P em uma cláusula C.

Utilizando o programa *smallest* apresentado e aplicando o operador de retirada de predicados, os seguintes mutantes podem ser gerados:

Mutante 1:

```
smallest([A],A).
smallest([A|B],N) :- smallest(B,N), N<A, !.
smallest([A| _ ], A).
```

Mutante 2:

```
smallest([A],A) :- !.
smallest([A|B],N) :- N<A, !.
smallest([A| _ ], A).
```

Mutante 3:

```
smallest([A],A) :- !.
smallest([A|B],N) :- smallest(B,N), !.
smallest([A| _ ], A).
```

Mutante 4:

```
smallest([A],A) :- !.
smallest([A|B],N) :- smallest(B,N), N<A.
smallest([A| _ ], A).
```

3.2.1.2 Troca de predicados(Swap Predicates)

Troca a ordem de predicados adjacentes P_1 e P_2 em uma cláusula C.

Utilizando o programa *smallest* apresentado e aplicando o operador de troca de predicados, os seguintes mutantes podem ser gerados:

Mutante 1:

```
smallest([A],A) :- !.
smallest([A|B],N) :- N<A, smallest(B,N), !.
smallest([A| _ ], A).
```

Mutante 2:

```
smallest([A],A) :- !.
smallest([A|B],N) :- smallest(B,N), !, N<A.
smallest([A| _ ], A).
```

3.2.1.3 Troca de conjunção por disjunção (Conjunction by Disjunction Replacement)

Troca a conjunção entre predicados P_1 e P_2 por uma disjunção em uma cláusula C.

Utilizando o programa *subset* apresentado e aplicando o operador de troca de conjunção por disjunção, os seguintes mutantes podem ser gerados:

Mutante 1:

```
subset(S,[H|T]):-subset(R,T), (S=R; S=[H|R]).
subset([],[]).
```

3.2.1.4 Troca de disjunção por conjunção (Disjunction by Conjunction Replacement)

Troca a disjunção entre predicados P_1 e P_2 por uma conjunção em uma cláusula C.

Utilizando o programa *least_num* apresentado e aplicando o operador de troca de disjunção por conjunção, os seguintes mutantes podem ser gerados:

Mutante 1:

```
least_num(X,[H|T]) :- least_num(Y,T), (H=<Y; X=H ; H>Y, X=Y).
```

Mutante 2:

```
least_num(X,[H|T]) :- least_num(Y,T), (H=<Y, X=H ; H>Y; X=Y).
```

3.2.1.5 Insere Cut (Insert Cut)

Insere um "cut" (!) entre os predicados P_1 e P_2 em uma cláusula C.

Utilizando o programa *smallest* apresentado e aplicando o operador de inserção de cut, os seguintes mutantes podem ser gerados:

Mutante 1:

```
smallest([A],A) :- !.
smallest([A|B],N) :- !, smallest(B,N), N<A, !.
smallest([A| _ ], A).
```

Mutante 2:

```
smallest([A],A) :- !.
smallest([A|B],N) :- smallest(B,N), !, N<A, !.
smallest([A| _ ], A).
```

Mutante 3:

```
smallest([A],A) :- !.
smallest([A|B],N) :- smallest(B,N), N<A, !.
smallest([A| _ ], A) :- !.
```

3.2.1.6 Remove Cut (Remove Cut)

Remove uma ocorrência do "cut" (!) em uma cláusula C.

Utilizando o programa *smallest* apresentado e aplicando o operador de remoção de cut, os seguintes mutantes podem ser gerados:

Mutante 1:

```
smallest([A],A).
smallest([A|B],N) :- smallest(B,N), N<A, !.
smallest([A| _ ], A).
```

Mutante 2:

```
smallest([A],A) :- !.
smallest([A|B],N) :- smallest(B,N), N<A.
smallest([A| _ ], A).
```

3.2.1.7 Permuta Cut (Permute Cut)

Permuta uma ocorrência do "cut" (!) em uma cláusula C.

Utilizando o programa *smallest* apresentado e aplicando o operador de permutação de cut, os seguintes mutantes podem ser gerados:

Mutante 1:

```
smallest([A],A) :- !.
smallest([A|B],N) :- !, smallest(B,N), N<A.
smallest([A| _ ], A).
```

Mutante 2:

```
smallest([A],A) :- !.
smallest([A|B],N) :- smallest(B,N), !, N<A.
smallest([A| _ ], A).
```

3.2.2 Mutação em Operadores (Operator Mutation)

3.2.2.1 Operador Aritmético (Arithmetic Operator Mutation)

Troca um operador aritmético por outro operador aritmético. Os operadores aritméticos manipulados são os seguintes: +, -, *, /.

Utilizando o programa *length* apresentado e aplicando o operador de mutação de operador aritmético, os seguintes mutantes podem ser gerados:

Mutante 1:

```
length([], 0).
length([A|B],N) :- length(B,M), N is M-1 .
```

Mutante 2:

```
length([], 0).
length([A|B],N) :- length(B,M), N is M*1 .
```

Mutante 3:

```
length([], 0).
length([A|B],N) :- length(B,M), N is M/1 .
```

3.2.2.2 Operador Relacional (Relational Operator Mutation)

Troca um operador relacional por outro operador relacional. Os operadores relacionais manipulados são os seguintes: <, >, ==, =, <= e >=.

Utilizando o programa *smallest* apresentado e aplicando o operador de mutação de operador relacional, os seguintes mutantes podem ser gerados:

Mutante 1:

```
smallest([A],A) :- !.
smallest([A|B],N) :- smallest(B,N), N>A, !.
smallest([A| _ ], A).
```

Mutante 2:

```
smallest([A],A) :- !.
smallest([A|B],N) :- smallest(B,N), N\== A, !.
smallest([A| _ ], A).
```

Mutante 3:

```
smallest([A],A) :- !.
smallest([A|B],N) :- smallest(B,N), N=A, !.
smallest([A| _ ], A).
```

Mutante 4:

```
smallest([A],A) :- !.
smallest([A|B],N) :- smallest(B,N), N<=A, !.
smallest([A| _ ], A).
```

Mutante 5:

```
smallest([A],A) :- !.
smallest([A|B],N) :- smallest(B,N), N>=A, !.
smallest([A| _ ], A).
```

3.2.3 Mutação em Variáveis (Variable Mutation)

3.2.3.1 Variável por Variável (Variable by Variable)

Troca uma variável em uma cláusula C por uma outra em C.

Utilizando o programa *member* apresentado e aplicando o operador de mutação de variável por variável, os seguintes mutantes podem ser gerados:

Mutante 1:

```
member(X,[X|_]).
member(X,[_|X]) :- member(X,T).
```

Mutante 2:

```
member(X,[X|_]).
member(X,[_|T]) :- member(X,X).
```

Mutante 3:

```
member(X,[X|_]).
member(T,[_|T]) :- member(X,T).
```

Mutante 4:

```
member(X,[X|_]).
member(X,[_|T]) :- member(T,T).
```

3.2.3.2 Variável por Variável Anônima(Variable by Anonymous Variable)

Troca uma variável em uma cláusula C por uma variável anônima de C, apenas na parte referente a chamada da cláusula, ou seja, até o ":-", ou se este não ocorrer, até o ponto final.

Utilizando o programa *member* apresentado e aplicando o operador de mutação de variável por variável anônima, os seguintes mutantes podem ser gerados:

Mutante 1:

```
member(_,[X|_]).
member(X,[_|T]) :- member(X,T).
```

Mutante 2:

```
member(X,[_|_]).
member(X,[_|T]) :- member(X,T).
```

Mutante 3:

```
member(X,[X| _]).
member(_,[_|T]) :- member(X,T).
```

Mutante 4:

```
member(X,[X| _]).
member(X,[_|_]) :- member(X,T).
```

3.2.3.3 Variável Anônima por Variável (Anonymous Variable by Variable)

Troca uma variável anônima em uma cláusula C por uma variável em C, apenas na parte referente a chamada da cláusula, ou seja, até o ":-", ou se este não ocorrer, até o ponto final.

Utilizando o programa *member* apresentado e aplicando o operador de mutação de variável anônima por variável, os seguintes mutantes podem ser gerados:

Mutante 1:

```
member(X,[X|X]).
member(X,[_|T]) :- member(X,T).
```

Mutante 2:

```
member(X,[X|_]).
member(X,[X|T]) :- member(X,T).
```

Mutante 3:

```
member(X,[X|_]).
member(X,[T|T]) :- member(X,T).
```

3.2.4 Mutação em Constantes (Constant Mutation)

3.2.4.1 Constante por Constante (Constant by Constant)

Troca uma constante em uma cláusula C por uma outra constante.

Utilizando o programa *mult* apresentado e aplicando o operador de mutação de constante por constante, os seguintes mutantes podem ser gerados:

Mutante 1:

```
mult([],[]).
mult([X|P],[N1,N2|Y]):-mult(P,Y), N1 is X+2, N2 is X+2.
```

Mutante 2:

```
mult([],[]).
mult([X|P],[N1,N2|Y]):-mult(P,Y), N1 is X+1, N2 is X+1.
```

3.2.4.2 Constante por Variável Anônima (Constant by Anonymous Variable)

Troca uma constante em uma cláusula C por uma variável anônima, apenas na parte referente a chamada da cláusula.

Utilizando o programa *length* apresentado e aplicando o operador de mutação de constante por variável anônima, o seguinte mutante pode ser gerado:

Mutante 1:

```
length([], _).
length([A|B],N) :- length(B,M), N is M+1 .
```

3.2.4.3 Variável Anônima por Constante (Anonymous Variable by Constant)

Troca uma variável anônima em uma cláusula C por uma constante, apenas na parte referente a chamada da cláusula.

Utilizando a cláusula *inicializa* e aplicando o operador de mutação de variável anônima por constante, o seguinte mutante pode ser gerado:

Mutante 1:

```
inicializa(0,0,0).
```

3.2.4.4 Constante por Variável (Constant by Variable)

Troca uma constante em uma cláusula C por uma variável.

Utilizando o programa *length* apresentado e aplicando o operador de mutação de constante por variável, os seguintes mutantes podem ser gerados:

Mutante 1:

```
length([], 0).
length([A|B],N) :- length(B,M), N is M+A.
```

Mutante 2:

```
length([], 0).
length([A|B],N) :- length(B,M), N is M+B.
```

Mutante 3:

```
length([], 0).
length([A|B],N) :- length(B,M), N is M+N.
```

Mutante 4:

```
length([], 0).
length([A|B],N) :- length(B,M), N is M+M.
```

3.2.4.5 Variável por Constante(Variable by Constant)

Troca uma variável em uma cláusula C por uma constante.

Utilizando o programa *length* apresentado e aplicando o operador de mutação de variável por constante, os seguintes mutantes podem ser gerados:

Mutante 1:

```
length([], 0).
length([1|B],N) :- length(B,M), N is M+1.
```

Mutante 2:

```
length([], 0).
length([A|1],N) :- length(B,M), N is M+1.
```

Mutante 3:

```
length([], 0).
length([A|B],1) :- length(B,M), N is M+1.
```

Mutante 4:

```
length([], 0).
length([A|B],N) :- length(1,M), N is M+1.
```

Mutante 5:

```
length([], 0).
length([A|B],N) :- length(B,1), N is M+1.
```

Mutante 6:

```
length([], 0).
length([A|B],N) :- length(B,M), 1 is M+1.
```

Mutante 7:

```
length([], 0).
length([A|B],N) :- length(B,M), N is 1+1.
```

3.3 Revelando defeitos com os operadores propostos

Abaixo segue um exemplo de como a ferramenta auxilia na descoberta do erro diretamente (*findteacher*) e indiretamente (*merge*), ou seja, o erro no programa não é descrito diretamente pelos operadores.

Programa *findteacher*

```
findteacher:-student(X),write("there exist students"), !,
            teacher(Y), !, write("teacher is ", Y).
```

Figura 3.1: Código Fonte do Programa *findteacher*

```
findteacher:-student(X),write("there exist students"),
             teacher(Y), !, write("teacher is ", Y).
```

Figura 3.2: Versão Incorreta do Programa *findteacher*

O programa findteacher, extraído de [18], é um exemplo de um erro que seria diretamente apontado pela ferramenta, através do operador de inclusão de cut. A Figura 3.1 mostra o programa incorreto enquanto que a Figura 3.2 mostra a versão incorreta. O erro está na falta de um cut, o que permitiria que o programa escrevesse várias vezes a frase "there exist students".

Programa *merge*

```
merge(A,[],A).
merge([],B,B).
merge([A|],[B|Rb],[A|M]) :-  A < B, merge(Ra,[B|Rb],M).
merge([A|Ra],[B|Rb],[A|M]) :-  A = B, merge(Ra,Rb,M).
merge([A|Ra],[B|Rb],[B|M]) :-  A > B, merge([A|Ra],Rb,M).
```

Figura 3.3: Código Fonte do Programa *merge*

```
merge(A,[],A).
merge([],B,B).
merge([A|Ra],[B|Rb],[A|M]) :-  A <= B, merge(Ra,[B|Rb],M).
merge([A|Ra],[B|Rb],[B|M]) :-  A > B, merge([A|Ra],Rb,M).
```

Figura 3.4: Versão Incorreta do Programa *merge*

Considere o programa *merge*, extraído de [1], na Figura 3.3 e sua versão incorreta na Figura 3.4. O programa une duas listas, formadas em ordem crescente, eliminando os elementos repetidos. A versão incorreta possui uma cláusula a menos, o que faz com que o programa não elimine os elementos repetidos. Um teste considerando todos os ramos não garante a descoberta do engano. Entretanto, será necessário um dado de teste que contenha elementos repetidos para matar os mutantes gerados utilizando-se o operador de mutação "Relational Operator". Esse dado revelará o defeito.

3.4 Considerações finais

O conjunto de operadores propostos nesse capítulo permite a utilização do critério Análise de Mutantes no teste de programas Prolog. O conjunto foi implementado pela ferramenta MutProlog, descrita no capítulo seguinte. Resultados sobre a avaliação do conjunto proposto são apresentados no Capítulo 5.

CAPÍTULO 4

DESCRIÇÃO DA FERRAMENTA MUTPROLOG

A ferramenta MutProlog foi desenvolvida para dar suporte ao teste de programas Prolog e implementa os operadores propostos no capítulo anterior. A MutProlog baseia-se na ferramenta Proteum, que auxilia o teste de programas em C [8], sem ter aproveitado o código fonte desta.

Para a análise de um programa pela ferramenta é necessário, além do código fonte, um arquivo de configuração. O código fonte a ser testado deverá estar sintaticamente correto e livre de erros de compilação e ter sido escrito em SWI Prolog. Esse Prolog foi escolhido por ser de domínio público e bastante utilizado no meio acadêmico. O arquivo de configuração contém algumas informações tais como o número de valores de entrada e saída, necessário para a execução do programa e a porcentagem que será aplicada para cada operador de mutação durante a geração dos mutantes.

MutProlog é formada por 4 componentes, que realizam a seguinte sequência de atividades: geração dos mutantes, criação dos scripts, execução dos mutantes e avaliação dos resultados. A Figura 4.1 mostra a sequência de execução dos componentes, a interação entre eles e suas entradas e saídas.

Os processos de geração dos mutantes e criação dos scripts são executados apenas uma vez. Já a execução dos mutantes (que engloba a tarefa da inclusão de dados de teste) e avaliação dos resultados pode ser executada várias vezes, enquanto o usuário achar necessário melhorar a cobertura dos seus casos de teste.

Toda a ferramenta foi desenvolvida na linguagem C sobre a plataforma Linux, sendo que cada componente é um executável em separado. Existe ainda o executável MutProlog que chama todos os componentes na sequência correta.

As próximas seções apresentam uma descrição desses componentes, bem como seus principais aspectos de implementação.

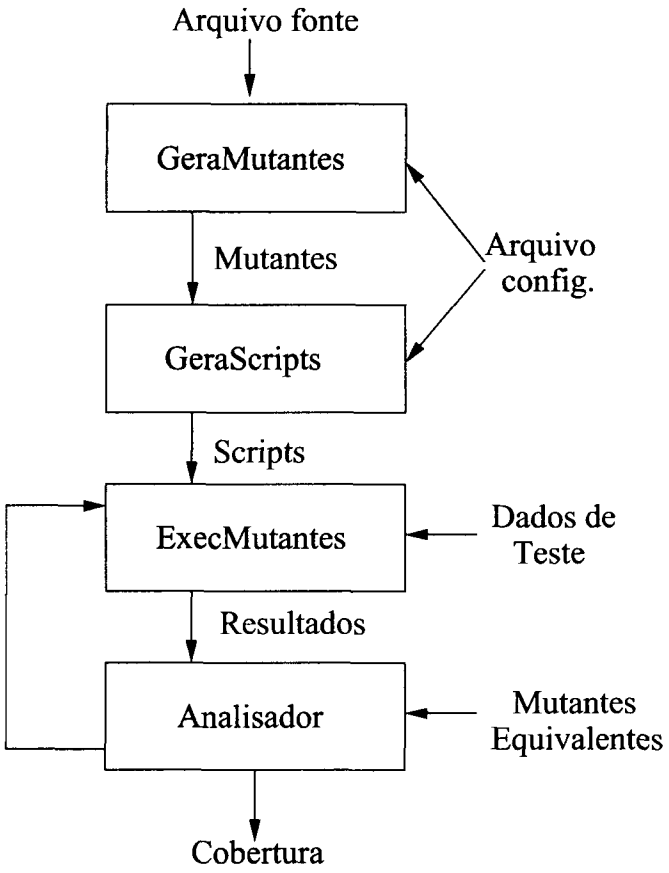

Figura 4.1: Componentes do MutProlog

4.1 GeraMutantes

Este componente é responsável por gerar os mutantes que serão utilizados durante o teste. As entradas deste componente são o programa fonte e o arquivo de configuração e a saída será um conjunto de programas em um diretório denominado Mutantes, dentro do diretório onde está o arquivo fonte do programa em teste. Todas as cláusulas presentes no arquivo fonte sofrerão mutações, mas cada uma é considerada em particular.

O arquivo de configuração fornece as porcentagens que serão aplicadas sobre cada operador de mutação individualmente. Por exemplo, para o programa *merge* da Figura 3.3, o arquivo de configuração da Figura 4.2 indica uma porcentagem de 50% para o operador "Insert Cut". Portanto, se para uma cláusula poderiam ser criados 4 mutantes, devido à porcentagem fornecida pelo usuário, serão criados apenas 2. A ordem da geração desses mutantes é detalhada na Seção 4.7.

```
input_variable=X
input_variable=Y
output_variable=Z
ClauseMutationPredicateDeletion=100
ClauseMutationSwapPredicates=100
ClauseMutationConjunctionByDisjunctionReplacement=100
ClauseMutationDisjunctionByConjunctionReplacement=100
ClauseMutationInsertCut=50
ClauseMutationRemoveCut=100
ClauseMutationPermuteCut=100
OperatorMutationArithmetic=100
OperatorMutationRelational=100
VariableMutationVariableByVariable=100
VariableMutationVariableByVariableAnonymous=100
VariableMutationVariableAnonymousByVariable=100
ConstantMutationConstantByConstantReplacement=100
ConstantMutationConstantByVariableAnonymous=100
ConstantMutationVariableAnonymousByConstant=100
ConstantMutationConstantByVariable=100
ConstantMutationVariableByConstant=100
```

Figura 4.2: Arquivo de Configuração para o Programa *merge*

4.2 GeraScripts

O módulo GeraScript é responsável por gerar os scripts de execução dos programas em Prolog, scripts estes que podem ser executados através da linha de comando. Para cada mutante gerado é montado um script (Prolog Script), que acaba "encapsulando" o mutante e permite que sejam passados os parâmetros normalmente, mostrando o resultado na saída padrão. O MutProlog pode, de posse destes scripts, executá-los diretamente a partir do módulo, que é escrito em linguagem C. Sem esses scripts, seria necessário executar o interpretador Prolog e carregar mutante por mutante, inviabilizando a automatização do processo de teste. Este módulo deve ser executado após o módulo GeraMutantes.

O módulo GeraScripts possui como parâmetros de entrada (linha de comando) o caminho do diretório em que se encontra a estrutura do programa Prolog (sem o nome do programa) e o nome do programa Prolog. Ao executar este módulo, para todos os programas que existirem no diretório Mutantes, um Prolog Script será montado e colocado no diretório Scripts.

Os parâmetros existentes no arquivo de configuração para o GeraScripts são os nomes

dos parâmetros de entrada e dos parâmetros de saída, que são usados para montar o script. O arquivo de configuração para o programa *merge* indica que o programa possui duas variáveis de entrada, X e Y, e produz uma saída Z.

4.3 ExecMutantes

O módulo ExecMutantes tem por finalidade executar todos os mutantes gerados pelo módulo Gerador de Mutantes com os dados de teste fornecidos pelo testador. Este módulo pode ser executado para adicionar dados de teste e/ou para executar os mutantes com os casos existentes. No primeiro caso, o programa em teste é executado e a saída obtida é armazenada. No segundo caso, as saídas do programa armazenadas serão comparadas com as saídas produzidas pelos mutantes.

O ExecMutantes pode ser executado quantas vezes forem necessárias, sendo que o Gerador de Mutantes e o Gerador de Scripts devem ter sido executados pelo menos uma vez para que existam mutantes que possam ser manipulados. A execução dos mutantes é realizada da seguinte maneira: para cada caso de teste, é criado um subdiretório em Resultados com o número do caso de teste e, dentro deste subdiretório, são armazenados os resultados de todos os mutantes.

4.4 Analisador

Este módulo tem por objetivo comparar o resultado do programa original com o resultado dos mutantes obtidos pelo ExecMutantes. A análise é baseada apenas no resultado gerado pela execução do mutante, sem levar em consideração os *warnings* que ocorrerem. Os mutantes que forem mortos tem o seu script e o arquivo fonte do mutante movidos para um diretório chamado MutantesMortos e os mutantes que continuarem vivos são mantidos nos mesmos lugares.

O resultado da avaliação é apresentado ao testador que poderá adicionar novos dados de teste necessários e também identificar os mutantes equivalentes que deverão ser descontados no cálculo da cobertura. Os mutantes marcados como equivalentes ou mortos

não serão executados pelo módulo ExecMutantes novamente.

4.5 Exemplo

A ferramenta MutProlog exige do usuário apenas dois arquivos: o código fonte do programa a ser testado e o arquivo de configuração, sendo a única exigência que esses arquivos sejam colocados nos diretórios Fonte e Config, respectivamente. Se, por exemplo, o arquivo em teste é o *uniao.pl*, deve ser criada uma estrutura de diretórios na qual o diretório raiz seria *uniao* e, dentro deste, um subdiretório Fonte (que conterá o arquivo uniao.pl) e o diretório Config (que conterá o arquivo uniao.cfg).

A partir deste ponto, o usuário pode executar o MutProlog ou cada componente individualmente. Para exemplificarmos todos os passos, iremos analisar a execução de componente por componente.

Para executar o módulo GeraMutantes, o usuário deve executar o seguinte comando: *./GeraMutantes ~ /TesteProlog uniao*, supondo que a estrutura para a execução do programa uniao esteja dentro do diretório TesteProlog. A execução do módulo GeraMutantes mostra a quantidade total de mutantes gerados, a quantidade gerada a partir de cada operador e cria um diretório Mutantes, dentro do diretório *uniao*, no qual os mutantes gerados são armazenados. Os mutantes gerados terão o seguinte padrão de nomenclatura: Mutante[NUMERO].pl, no qual NUMERO é a ordem na qual o mutante foi gerado. Se o programa *uniao* gerar 20 mutantes, no diretório *uniao/Mutantes* teremos 20 arquivos, cujos nomes serão Mutante1.pl, Mutante2.pl, ... , Mutante20.pl.

Após a geração dos mutantes, o módulo a ser executado é o GeraScripts. A execução deste componente, utilizando o programa *uniao*, deve ser feito da seguinte forma: o usuário *./GeraScripts ~ /TesteProlog uniao*. Na execução desse módulo é criado o diretório Scripts dentro do diretório *uniao*, no qual todos os scripts gerados são armazenados. Para cada mutante existente um script é gerado. Seguindo o exemplo do programa *uniao* ter gerado 20 mutantes, no diretório ~ *uniao/Scripts* teremos 20 arquivos, cujos nomes serão Script1.sh, Script2.sh, ... , Script20.sh.

Os dois módulos executados até este ponto, o GeraMutantes e o GeraScripts, são

executados apenas uma vez para cada programa a ser testado. Os próximos 2 módulos, o ExecMutantes e o Analisador, permitem ao usuário escolher quantas vezes os mutantes ainda vivos devem ser executados. O objetivo disto é permitir a análise dos casos de teste executados, verificando se mais casos são necessários para que uma melhor cobertura seja obtida.

O módulo ExecMutantes possui duas funcionalidades: permite a entrada dos casos de teste e executa todos os mutantes com os casos disponíveis. O usuário pode incluir os casos de teste através do módulo ou diretamente no diretório CasosDeTeste. Após a inclusão dos casos de teste, o ExecMutantes executa os mutantes e coloca os resultados obtidos dentro do subdiretório CasosDeTeste. A organização no diretório CasosDeTeste é a seguinte: para cada caso de teste é criado um novo diretório, no qual são armazenados os resultados de todos os mutantes.

O último componente a ser executado é o Analisador, que verifica o resultado gerado pelo programa fonte em relação aos mutantes. Caso o resultado seja diferente, o mutante e o seu script são retirados do diretório Mutantes e colocados no diretório MutantesMortos.

4.6 Implementação

A ferramenta foi desenvolvida na linguagem C sobre a plataforma Linux, sendo que cada componente é um executável separado. Como verificado, MutProlog trabalha sobre uma estrutura de diretórios e arquivos, arquivos estes que contêm todas as informações relativas ao programa em teste, como mutantes mortos e vivos, casos de teste, scripts, etc.

O módulo que exigiu um maior esforço de programação foi o GeraMutantes, por toda a complexidade que envolve a geração das alterações sintáticas no programa original. Inicialmente foi desenvolvida uma biblioteca de funções que seriam comuns a todos os operadores, com o objetivo de minimizar a replicação de código e facilitar a atividade de teste da ferramenta. Sobre esta camada de funções genéricas, foram construídos os operadores, que possuem uma interface comum para facilitar sua chamada. Esses operadores percorrem o arquivo fonte, obtêm as cláusulas e, conforme sua funcionalidade, utilizam a biblioteca para montar os mutantes. No nível mais alto deste componente, existe o

arquivo que é executado, o GeraMutantes.c, cuja funcionalidade é percorrer o arquivo de configuração e executar os operadores, conforme indicado pelo usuário. Os outros 3 componentes possuem apenas um arquivo fonte e utilizam algumas chamadas de sistema para executar a manipulação dos arquivos.

Outro detalhe a ser comentado é a ordem de geração dos mutantes quando a porcentagem não é 100: neste caso a ordem de geração é sempre da esquerda para a direita. No caso de mutação em relação a variáveis e constantes, a ordem também é da esquerda para a direita em relação à ordem em que são encontradas. No Capítulo 3, foram apresentados todos os mutantes possíveis para os programas e operadores apresentados, o equivalente a considerar 100%.

4.7 Exemplo de saída da ferramenta

Abaixo temos um log da execução da MutProlog para o programa *merge*.

```
Mutantes gerados:
- ClauseMutationPredicateDeletion: 6
- ClauseMutationSwapPredicates: 3
- ClauseMutationInsertCut: 9
- OperatorMutationRelationalOperatorMutation: 15
- VariableMutationVariableByVariable: 140
- VariableMutationVariableByVariableAnonymous: 13
Total de mutantes gerados: 186

Execução:
Casos de teste realizados:

Teste 1:
Entrada: Lista 1: [1,2] - Lista 2: [] - Saída do programa original: [1,2]
Mutantes Mortos: 89 - Mutantes Vivos: 97
```

Teste 2:

Entrada: Lista 1: [] - Lista 2: [1,2] - Saída do programa original: [1,2]

Mutantes Mortos: 62 - Mutantes Vivos: 35

Teste 3:

Entrada: Lista 1: [1,2] - Lista 2: [1,2] - Saída do programa original: [1,2]

Mutantes Mortos: 17 - Mutantes Vivos: 45

Teste 4:

Entrada: Lista 1: [2,3] - Lista 2: [1,2] - Saída do programa original: [1,2,3]

Mutantes Mortos: 30 - Mutantes Vivos: 15

Teste 5:

Entrada: Lista 1: [0,4] - Lista 2: [1,2,3] - Saída do programa original:
[0,1,2,3,4]

Mutantes Mortos: 0 - Mutantes Vivos: 15

Teste 6:

Entrada: Lista 1: [1,3,5] - Lista 2: [2,4,6] - Saída do programa original:
[1,2,3,4,5,6]

Mutantes Mortos: 0 - Mutantes Vivos: 15

Mutantes Vivos:

5, 7, 8, 9, 10, 11, 12, 13, 14, 16, 18, 20, 56, 58, 102

4.8 Considerações finais

Neste capítulo a ferramenta MutProlog foi apresentada através de seus componentes e da interação entre eles. Pode-se perceber que MutProlog implementa os operadores propostos no capítulo 3, permitindo assim o teste de programas em Prolog através do Critério Análise de Mutantes. No próximo capítulo teremos o resultado dos experimentos da utilização da MutProlog seguido da conclusão deste trabalho.

CAPÍTULO 5

EXPERIMENTOS

Este capítulo descreve o experimento realizado com a MutProlog. O objetivo principal desse experimento, além de validar a implementação da ferramenta, foi realizar uma análise dos operadores propostos e compará-los com a abordagem estrutural[8], em termos de custo, eficácia e *strenght*. Não foi possível a comparação com a abordagem que utiliza Programação Lógica Indutiva (ILP-Inductive Logic Programming) ou Programação Genética, descritas no Capítulo 2, pela não existência de uma ferramenta de teste para apoiar essas abordagens, dificultando a aplicação das mesmas.

5.1 Descrição

O experimento utilizou quatro programas(*elem_rep*, *num_ap*, *ord_sel* e *merge*[1]), cujos códigos fonte encontram-se no Apêndice A. Os seguintes passos foram seguidos para cada programa:

1. geração dos arquivos de configuração. Todos os operadores foram utilizados com os parâmetros de 100%.
2. geração dos mutantes.
3. geração de dados de teste e submissão à ferramenta.
4. execução dos mutantes e avaliação dos resultados.
5. geração de dados de teste adicionais para os mutantes vivos e determinação dos mutantes equivalentes.
6. execução dos mutantes com os novos casos de teste.
7. repetição dos dois últimos passos até que todos os mutantes não equivalentes tenham sido mortos e obtenção da cobertura final e dos conjuntos T_{mp} MutProlog adequados.

Tabela 5.1: Número de Ramos Requeridos e Não executáveis

Programa	Nro ramos requeridos	Não executáveis
elem_rep	17	0
num_ap	14	0
ord_sel	10	0
merge	26	0
Total	67	0

Com a execução desses passos, a Tabela 5.2 foi obtida. Esta tabela mostra, por exemplo, que para o programa *merge* foram gerados 186 mutantes, 171 foram mortos com 4 casos de teste (casos de teste que realmente mataram algum mutante) e 15 são equivalentes (cerca de 8%). O número de mutantes gerados e equivalentes para cada operador e cada programa são apresentados na Tabela 5.3.

Após a obtenção dos conjuntos T_{mp} foram realizados os seguintes passos para cada programa com objetivo de permitir a comparação com a abordagem estrutural:

- Geração dos *P-flowgraphs* para os programas e dos elementos requeridos pelo critério todos-ramos.

- Avaliação da cobertura dos conjuntos T_{mp} (MutProlog adequados) para os critérios estruturais.

- Identificação dos elementos estruturais não executáveis. A Tabela 5.1 apresenta o número total de elementos requeridos e não executáveis para cada $T_{mp}i$.

- Obtenção da cobertura dos conjuntos $T_{mp}i$ para o critério estrutural dos casos de teste realmente efetivos. Esse resultado é apresentado na Tabela 5.4. Como se pode observar, para todos os programas a cobertura do conjunto T_{mp} foi sempre 100%, obtendo-se, dessa maneira, para cada programa, conjuntos adequados ao critério estrutural T_ci.

- Obtenção do escore de mutação obtido na MutProlog com os conjuntos T_ci. Os escores obtidos estão na Tabela 5.5.

Tabela 5.2: Mutantes Gerados, Equivalentes e Casos de Teste Necessários

Programa	Mutantes Gerados	Equivalentes	Nro. de Casos de Teste
elem_rep	314	21 (6,68%)	3
num_ap	208	18 (8,65%)	3
ord_sel	187	9 (4,81%)	3
merge	186	15 (8,06%)	4
Total	895	63 (7,03%)	13

5.2 Análise dos Resultados

Os números observados na Tabela 5.2 são relativamente pequenos quando comparados com o número de mutantes geralmente gerados para programas escritos em linguagens procedurais. Isso se deve ao fato de que os programas Prolog são de tamanho menor, no número de linhas e variáveis utilizadas.

A porcentagem de programas equivalentes encontrada, em torno de 7%, é baixa, o que mostra que o conjunto de operadores proposto gerou um número pequeno de programas equivalentes. Isso é muito importante, pois a determinação manual desses programas consome bastante tempo e esforço na atividade de teste. O número de casos de teste necessários para cada programa também é pequeno, isso devido ao tipo de programas utilizados (simples).

Com relação à Tabela 5.3, observa-se que a maior parte dos mutantes foi gerada por operadores do grupo mutação em variáveis e em segundo lugar por operadores do grupo mutação em cláusulas. Isso se deve ao fato de que em Prolog os operadores são menos utilizados do que em linguagens tradicionais. As listas são mais importantes. Nesse sentido, verifica-se a importância dos operadores do grupo mutação em variáveis, pois muitas dessas listas utilizam-se de variáveis (anônimas ou não).

Apesar de se ter utilizado uma porcentagem de 100% para a aplicação dos operadores de mutação nem todos os operadores foram utilizados para a amostra de programas usados. Nenhum programa, por exemplo, apresenta disjunção ou conjunção. Novos estudos deverão ser realizados para avaliar esses operadores.

O número de programas equivalentes encontrados foi bastante pequeno, ou seja, a porcentagem encontrada foi bem menor do que em programas tradicionais. Observa-se que

Tabela 5.3: Número de Mutantes Gerados e Equivalentes Por Operador

	elem_rep		num_ap		ord_sel		merge	
Operador	Gerados	Equiv.	Gerados	Equiv.	Gerados	Equiv.	Gerados	Equiv.
Clause Mutation								
Predicate Deletion	11	4	10	2	6	1	6	1
Swap Predicates	5	2	4		3	1	3	3
Conjunc. by Disjunc.								
Disjunc. by Conjunc.								
Insert Cut	13	10	17	16	9	7	9	7
Remove Cut			1					
Permut Cut			3					
Operator Mutation								
Aritmetic Operator	3		3					
Relational Operator	5	1	5		15		15	1
Variable Mutation								
Variable by Variable	201	4	130		136		140	3
Variable by Anonymous	25		17		18		13	
Anonymous by Variable	3		3					
Constant Mutation								
Constant by Constant								
Constant by Anonymous			1					
Anonymous by Constant			1					
Constant by Variable	15		4					
Variable by Constant	33		9					
Total	314	21	208	18	187	9	186	15

Tabela 5.4: Cobertura MutProlog X Fluxo de Controle

Programa	Nro. Casos de Teste	Cobertura Fluxo Controle
elem_rep	1	100%
num_ap	1	100%
ord_sel	1	100%
merge	3	100%
Total	6	100%

Tabela 5.5: Cobertura utilizando Fluxo de Controle X MutProlog

Programa	Nro. Casos de Teste	Cobertura MutProlog
elem_rep	1	79,8%
num_ap	1	93,75%
ord_sel	1	92,5%
merge	3	97,07%
Total	6	90,25%

o operador "Insert Cut" foi o que mais gerou equivalência em número e em porcentagem. Talvez esse operador deva receber uma atenção mais detalhada por parte do testador, que poderá não exigir todas as combinações possíveis, ou a ferramenta MutProlog deverá ser alterada com relação a inserir "cuts".

Durante a realização desse experimento também observou-se que seria interessante acrescentar um operador para realizar mutações em listas utilizando o operador "|". Pretende-se estudar melhor os tipos de mutações que seriam possíveis e assim refinar o conjunto proposto e implementar esse operador na ferramenta.

O número de casos de teste necessários para matar os mutantes gerados pela MutProlog, conforme verificado, pode ser considerado pequeno. Mas, deve-se levar em conta que o número de casos de teste anotados são os casos de teste efetivos, aqueles que realmente matam os mutantes.

5.3 Comparação com a abordagem estrutural

5.3.1 Custo

Como observado em outros trabalhos da literatura, o custo de aplicação do critério análise de mutantes foi 2 vezes maior que o custo da aplicação do critério estrutural. Em programas tradicionais esse custo pode ser de até 3 vezes mais, por isso, o custo obtido foi menor que o esperado.

5.3.2 *Strenght*

Observa-se que os dados gerados pelo teste de mutação cobriram 100% dos ramos executáveis do grafo de fluxo de controle, e os dados gerados para satisfazer o critério estrutural mataram, na média, 90% dos mutantes não equivalentes. O conjunto todos-ramos adequado não garantiu a morte de 10% dos mutantes não equivalentes gerados pelo MutProlog, conseqüentemente não garantindo que os defeitos descritos pelos operadores que geraram esses mutantes sejam revelados.

Tabela 5.6: Número de Erros Revelados pelos Conjuntos

Programas	Num. Erros	Num. Erros revelados pelo $T_c i$	Num. Erros revelados pelo $T_{mp} i$
elem_rep	5	4 (80%)	5 (100%)
num_ap	5	5 (100%)	5 (100%)
ord_sel	5	4 (100%)	4 (80%)
merge	5	5 (60%)	5 (100%)
Total	20	18 (90%)	19 (95%)

5.3.3 Eficácia

Com o objetivo de se avaliar a eficácia de ambos os critérios foram geradas versões incorretas para os programas. Essas versões foram geradas introduzindo-se 1 (um) erro ou mais em cada programa, diferentemente dos operadores de mutação do MutProlog. A Tabela 5.6 apresenta o número total de erros inseridos aleatoriamente em cada programa. Essas versões incorretas também são apresentadas no Apêndice A. As versões incorretas foram executadas pelos conjuntos $T_c i$ e $T_{mp} i$. O número de erros revelado com cada conjunto também pode ser visualizado na Tabela 5.6. O conjunto $T_{mp} i$ revelou 1 (um) erro (5%) a mais que o conjunto $T_c i$.

5.4 Considerações finais

O experimento descrito nesta seção mostrou a aplicabilidade dos operadores de mutação propostos. A porcentagem de mutantes gerados e a porcentagem de mutantes equivalentes encontrada foi menor que em programas tradicionais, isso devido ao tamanho dos programas Prolog que são em geral menores.

Observou-se a necessidade de se definir um novo operador de mutação para o operador Prolog "|" de listas e de se estudar o operador Insere Cut, que foi o que mais gerou mutantes equivalentes.

Com relação à comparação com a abordagem estrutural os resultados também são similares aos obtidos com programas tradicionais. O critério análise de mutantes requer um número maior de casos de teste que o critério todos-ramos e obtiveram-se indícios de uma maior eficácia. Os resultados com relação ao *strenght* também apontam para a relação empírica, mostrando que é mais difícil satisfazer o critério análise de mutantes do que o critério estrutural.

CAPÍTULO 6

CONCLUSÃO

Este trabalho abordou a utilização do critério Análise de Mutantes no teste de programas Prolog, visto que na literatura poucos são os autores que procuram estender, nesse contexto, os critérios e ferramentas de teste utilizados para programa tradicionais.

Critérios de teste específicos para programas Prolog, além de auxiliar a seleção de dados de teste mais eficazes também permitem a quantificação da atividade de teste, oferecendo medidas de cobertura, de avaliação e comparação de conjuntos de dados de teste.

O trabalho introduziu um conjunto de operadores de mutação baseado em características particulares dessa linguagem e numa classificação dos principais defeitos encontrados em programas Prolog. A utilização dos operadores permite a descoberta não somente de defeitos descritos por cada classe mas também de combinações de defeitos.

Uma ferramenta de teste de mutantes, chamada MutProlog, foi descrita. A ferramenta permite a aplicação prática dos operadores propostos, gerando e executando os mutantes automaticamente, além de realizar o cálculo da cobertura.

Um experimento foi conduzido com alguns programas encontrados em artigos da literatura e realizada uma avaliação inicial dos operadores. Os resultados indicam a aplicabilidade do conjunto proposto. O número de dados de teste necessários não tende a crescer proporcionalmente ao número de mutantes gerados. A porcentagem de mutantes equivalentes encontrada é baixa, o que contribui para diminuir o esforço do teste.

Os resultados obtidos com uma comparação com o critério estrutural, todos-ramos, são os esperados, por serem similares aos encontrados em experimentos com programas tradicionais. O conjunto de dados de teste adequado ao conjunto de operadores proposto foi capaz de cobrir todos os elementos executáveis requeridos pelo critério estrutural. Sendo que o contrário não foi possível, 10% dos mutantes gerados não puderam ser mortos

pelo conjunto de teste todos-ramos adequado. Esse fato traz indícios de que a aplicação dos operadores propostos implica em um maior número de defeitos revelados, comprovado pelo estudo da eficácia realizado. Cabe ao testador, de acordo com a confiabilidade requerida para o programa em teste, a escolha de um critério mais custoso.

6.1 Trabalhos futuros

Pretende-se, num segundo passo, realizar estudos para permitir o estabelecimento de operadores essenciais de mutação para a linguagem Prolog. Conjuntos de operadores seriam estabelecidos de acordo com o tipo de aplicação e características do programa em teste. Isso tem sido realizado com sucesso para diminuir o custos do teste de mutantes em programas tradicionais.

Novos estudos com outros programas também deverão ser conduzidos para permitir o refinamento dos operadores propostos e o fator eficácia deverá ser explorado futuramente com programas contendo erros naturais. Um operador de mutação para o operador de listas "|" deverá ser implementado, bem como operadores que possam testar erros de *backtracking* intercláusulas.

Algumas funcionalidades deverão ser acrescentadas à ferramenta MutProlog, tais como auxílio à determinação de mutantes equivalentes, habilitação de casos de teste, produção de relatórios e controle automático de mutantes anômalos. Uma interface gráfica para visualização dos mutantes e da cobertura deverá ser construída.

BIBLIOGRAFIA

[1] F. Bergadano e D. Gunetti. *Inductive Logic Programming: From Machine Learning to Software Engineering.* The MIT Press, 1996.

[2] P. Boeck e B. Charlier. Static type analysis of prolog procedures for ensuring correctness. *International Workshop PLILP*, páginas 223–237. Spring-Verlag, 1990.

[3] T.A. Budd e D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, Vol. 18(1):31–45, Novembro de 1982.

[4] N. Choquet. Test data generation using a prolog with constraints. *Proc. of the Workshop on Software Testing*, páginas 132–141. Computer Science Press, Banff - Canada, Julho de 1986.

[5] W.M. Craft. *Detecting Equivalents Mutants Using Compiler Optimization.* Master Thesis, Department of Computer Science, Clemson University, Clemson-SC, 1989.

[6] R.A. De Millo, D.C. Gwind, e K.N. King. An extented overview of the mothra software testing environment. *Proc. of the Second Workshop on Software Testing, Verification and Analysis*, páginas 142–151. Computer Science Press, Banff - Canada, Julho de 1988.

[7] R.A. De Millo, R.J. Lipton, e F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, Vol. C-11:34–41, Abril de 1978.

[8] M. E. Delamaro e J.C. Maldonado. A tool for the assesment fo test adequacy for c programs. *Proceedings of the Conference on Performability in Computing Systems, 79-95.* East Brunswick, New Jersey, USA, Julho de 1996.

[9] M.E. Delamaro. *Proteum: Um ambiente de teste baseado na Análise de Mutantes.* Master Thesis, ICMSC-USP-São Carlos, São Carlos - SP, Brasil, Março de 1993.

[10] R. Denney. Test case generation form prolog-based specifications. *IEEE Software, pgs 49-57*, Março de 1991.

[11] M.C.P.F. Emer e S.R. Vergilio. *Selection and Evaluation of Test Data Based on Genetic Programming.* Software Quality Journal, vol 11, num 2, Junho de 2003.

[12] F.G. Frankl. *The use of Data Flow Information for the Selection and Evaluation of Software Test Data.* PhD Thesis, Department of Computer Science, New York University, New York, U.S.A., Outubro de 1987.

[13] F.G. Frankl e E.J. Weyuker. Data flow testing in the presence of unexecutable paths. *Proceedings of the Workshop on Software Testing*, páginas 4–13. Computer Science Press, Banff - Canada, Julho de 1986.

[14] M.M. Gorlick, C.F. Kesselman, D.A. Marotta, e S Parker. Mockingbird: A logical methodology for testing. *Journal of Logic Programming*, (8):95–119, 1990.

[15] D. Hoffman e P. Strooper. Automated module testing in prolog. *iEEE Transactions on Software Engineering*, 17(9):934–943, 1991.

[16] W.E. Howden. *Functional Program Testing and Analysis.* McGrawHill, USA, 1987.

[17] IEEE. *IEEE Standard Glossary of Software Engineering Terminology.* New York, 1990.

[18] G.B. Luo, B. Sarikaya, e M. Boyer. Control-flow based testing of prolog programs. páginas 104–113, Março de 1992.

[19] J.C. Maldonado. *Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software.* DCA/FEEC/Unicamp, Campinas - SP, Brasil, Julho de 1991.

[20] J.C. Maldonado, S.R. Vergilio, M.L. Chaim, e M. Jino. Critérios potenciais usos: Análise de aplicação de um benchmark. *VI Simpósio Brasileiro de Engenharia de Software*, páginas 357–371. Gramado-RS, Brazil, Novembro de 1992.

[21] A.P Mathur e W.E. Wong. An empirical comparison of data flow and mutation based test adequacy criteria. *The Journal of Software Testing, Verification an Reliability*, Vol. 4(1):9–31, Março de 1994.

[22] L.M. Pereira. Rational debugging in logic programming. *Third International Conference on Logic Programming*, páginas 203–210. Lectures Notes on Computer Science, 1986.

[23] L. Plumer. Termination proofs for logic programs. *Lectures Notes in Artificial Intelligence*. Spring-Verlag, 1990.

[24] R.B. Pressman. *Engenharia de Software*. Makron Books, terceira edition, 1995.

[25] S. Rapps e E.J. Weyuker. Data flow analysis techniques for test data selection. *Proceedings of International Conference on Software Engineering*. Toquio - Japão, Setembro de 1982.

[26] S. Rapps e E.J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, Abril de 1985.

[27] S.R. Vergilio. *Critérios Restritos de Teste de Software: Uma Contribuição para Gerar Dados de Teste mais Eficazes*. DCA/FEEC/Unicamp, Campinas - SP, Brasil, Julho de 1997.

[28] E.J. Weyuker e R.G. Weiss, S.N.and Hamlet. Comparison of program testing strategies. *Proc. of the Fourth Symposium on Software Testing, Analysis and Verification, 154-164*. ACM Press, Victoria - Canada, 1991.

[29] W.E. Wong. *On Mutation and Data Flow*. PhD Thesis, Department of Computer Science, Purdue University, West Lafayette-IN, USA, Dezembro de 1993.

APÊNDICE A - PROGRAMAS UTILIZADOS NO EXPERIMENTO

Os códigos dos programas são os seguintes:

1. *elem_rep*

Mostra os elementos repetidos:

```
elementos_repetidos([],[]).
elementos_repetidos([X|Resto],[X|R]):-
        procurar_tirar(X,[X|Resto],Resto2,NumOcc),
        NumOcc > 1,
        elementos_repetidos(Resto2,R).
elementos_repetidos([_|Resto],R):-
        elementos_repetidos(Resto,R).


procurar_tirar(X,Lista,Res,N):-
        pt(X,Lista,Res,0,N).


pt(_,[],[],N,N).
pt(X,[X|Resto],Res,Acum,N):-
        Acum2 is Acum+1,
        pt(X,Resto,Res,Acum2,N).
pt(X,[Y|Resto],[Y|Res],Acum,N):-
        pt(X,Resto,Res,Acum,N).
```

Versões incorretas utilizadas:

1. Retirada a primeira cláusula de elementos_repetidos.
2. Retirada a última cláusula de elementos_repetidos.

3. Incluída outra condição de parada em elementos_repetidos com os parâmetros [ ] e _.

4. Retirada a última cláusula "pt".

5. Alterada a cláusula procurar_tirar: passa 1 (um) ao invés de 0 (zero).

2. *num_ap*

Número de vezes que um programa aparece na lista:

```
num_ap(Lista,Res):-
        num_ap2(Lista,[],Res).

num_ap2([],Res,Res).
num_ap2([X|Resto],L,Res):-
        not(member([X,_],L)),!,
        num_occ(X,[X|Resto],N),
        num_ap2(Resto,[[X,N]|L],Res).
num_ap2([_|Resto],L,Res):-
        num_ap2(Resto,L,Res).

num_occ(_,[],0).
num_occ(X,[X|Resto],Num):-
        num_occ(X,Resto,N),
        Num is N+1.
num_occ(X,[Y|Resto],Num):-
        X \== Y,
        num_occ(X,Resto,Num).
```

Versões incorretas utilizadas:

1. Retirada a primeira cláusula de num_ap2.

2. Retirada a última linha da terceira cláusula de num_occ.

3. num_ap passa _ ao invés de [ ].

4. Terceira clásula de num_ap passa Resto e _ ao invés de _ e Resto

5. Retirada a última cláusula de num_occ.

3. *ord_sel*

Ordena a lista de entrada:

```
ord_selecao([],[]).
ord_selecao([X|Resto],[Menor|L]):-
        selecionar_menor(X,Resto,Menor,Resto2),
        ord_selecao(Resto2,L).

selecionar_menor(X,[],X,[]).
selecionar_menor(X,[Y|Resto],Menor,[X|Resto2]):-
        Y < X,
selecionar_menor(Y,Resto,Menor,Resto2).
selecionar_menor(X,[Y|Resto],Menor,[Y|Resto2]):-
        Y >= X,
        selecionar_menor(X,Resto,Menor,Resto2).
```

Versões incorretas utilizadas:

1. Alterada a última cláusula de selecionar_menor: trocado Resto e Resto2.

2. Mesmo que no item acima mas na segunda cláusula de selecionar_menor.

3. Segunda cláusula de selecionar_menor troca X e Y na parte que recebe os parâmetros.

4. Retirada a primeira cláusula de selecionar_menor.

5. Alterada a segunda cláusula de selecionar_menor: troca Y por X e Resto2 por Resto na chamada do último método.

4. *merge*

Une as listas de entrada:

```
merge3([],B,B).
merge3(A,[],A).
merge3([A|Ra],[B|Rb],[A|M]):- A < B, merge3(Ra,[B|Rb],M).
merge3([A|Ra],[B|Rb],M):- A = B, merge3(Ra,[B|Rb],M).
merge3([A|Ra],[B|Rb],[B|M]):- A > B, merge3([A|Ra],Rb,M).
```

Versões incorretas utilizadas:

1. Tira recursão da terceira e quinta cláusulas.
2. Passa [ ] no segundo parâmetro da quinta cláusula.
3. Na quarta cláusula passa [ ] ao invés de Ra.
4. Tirados os dois predicados da última cláusula.
5. Passa [ ] no primeiro e segundo parâmetros da terceira e quinta cláusulas.