Definição de Operadores de Mutação para o Teste de Programas Prolog

Silvia Regina Vergilio silvia@inf.ufpr.br

Rudá Sumé Tente de Moura ruda@conectiva.com.br

Departamento de Informática
Universidade Federal do Paraná - UFPR
CP: 19081, 81531-970, Curitiba, PR

Resumo

Teste é uma das fases da engenharia de software que garante a qualidade e a confiabilidade do software desenvolvido. O critério de teste, baseado em erros, *Análise de Mutantes*, tem se mostrado como um dos mais efetivos para a detecção de erros em um programa, pois utiliza informação dos erros cometidos pelos próprios programadores para estabelecer o teste. Algumas ferramentas de testes, apoiodas no critério Análise de Mutantes, foram desenvolvidas e são utilizadas para o teste de programas que utilizam o paradigma da programação imperativa (C e Fortran). Entretanto, a idéia de aplicar o teste baseado em erros para programas escritos utilizando outros paradigmas de programação é bastante promissora. Este trabalho define um conjunto de operadores de mutação para a linguagem Prolog. O conjunto proposto permite a aplicação do critério Análise de Mutantes para o teste de programas Prolog e a implementação de uma ferramenta que poderá avaliar a eficácia do critério e o refinamento do conjunto proposto.

1 Introdução

A atividade de teste é uma das mais importantes fases do desenvolvimento de um software. Segundo Pressman [Pre97], esta fase corresponde entre 30% e 40% do custo total do software, elevando-se ainda mais quando o fator "risco humano" é considerado em um projeto, e uma alta confiabilidade é requerida.

Myers [Mye79] define a atividade de teste como o processo de executar um programa com o objetivo de encontrar erros. Por esta interpretação, um teste nunca poderá demonstrar que um programa não contém erros, apenas mostrará que eles estão presentes [DDH72]. Indiretamente, a atividade de teste serve para estabelecer que um software funciona de acordo com a sua especificação, aumentando sua confiabilidade e qualidade.

Diversas técnicas existem para selecionar bons dados de testes, estas técnicas podem ser classificadas da seguinte maneira: **técnica funcional** – nenhum conhecimento a priori da estrutura interna do programa é conhecido portanto, dados de testes são derivados a partir da especificação funcional do programa; **técnica estrutural** – os dados de teste são derivados através de uma análise da estrutura interna do programa. Detalhes como: complexidade ciclomática [McC76, Pre97], fluxo de controle [TLK92] e fluxo de dados [RW85, Mal91, Ver97] são utilizados como guia para algum critério de teste estrutural; e **técnica baseada em erros** – os dados de teste são derivados utilizando os tipos de erros mais comuns que ocorrem no processo de desenvolvimento de um software, de maneira a introduzir estes erros em um programa e verificar a presença ou ausência deste erro no programa em teste [DLS78].

As técnicas de teste estão associadas a critérios. Um critério de teste é um predicado a ser satisfeito para se considerar a atividade de teste, pode ser utilizado para avaliar um dado conjunto de casos de teste, oferecendo medidas de cobertura, bem como fornecer diretrizes que auxiliam a geração do dado de teste.

Entre os diversos critérios propostos para a atividade de teste, o critério Análise de mutantes, baseado em erros, tem despertado bastante interesse e se mostrado bastante efetivo em detectar erros. A Análise de Mutantes requer a criação de um conjunto de programas semelhantes ao programa original em teste (mutantes), mas que possuem pequenas diferenças sintáticas. O objetivo é encontrar um conjunto de casos de teste capaz de revelar diferenças entre o programa em teste e seus mutantes.

Para possibilitar a utilização prática deste 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, principalmente Fortran e C, não permitem a aplicação do critério em outros paradigmas de programação. Em particular, observa-se a pouca existência de ferramentas de testes para programas lógicos e principalmente para aqueles escritos na linguagem Prolog.

Bergadano e Gunetti [BG96a, BG96b] propõem a utilização da *Programação Lógica Indutiva* (ILP)[1] [MdR94] para auxiliar a atividade de teste. Conceitualmente, a ILP está preocupada com o problema de aprender programas lógicos a partir de exemplos de entrada e saída. A partir dos programas aprendidos os casos de teste são gerados a fim de diferenciá-los do programa original. Esta técnica difere do critério Análise de Mutantes pois os programas aprendidos não diferem do programa original apenas por mutações simples e não são utilizados operadores de mutação. Os operadores determinam as mudanças sintáticas no programa em teste para a geração de um programa mutante e descrevem erros em geral cometidos pelos programadores.

No entanto, o trabalho de Bergadano e Gunetti mostra como promissora a idéia de realizar teste baseado em erros para programas Prolog. Esse fato, aliado a inexistência de ferramentas como já citado acima, constituem motivações para a realização do trabalho aqui descrito, cujo objetivo é definir operadores de mutação para a linguagem Prolog e mostrar que o critério Análise de Mutantes pode auxiliar o teste de programas escritos nessa linguagem.

Primeiramente na Seção 2 será apresentada uma breve descrição do critério Análise de Mutantes. A Seção 3 contém uma proposta de operadores de mutação para a linguagem Prolog com exemplos. A Seção 4 apresenta um exemplo no qual a aplicação do critério Análise de Mutantes e dos operadores definidos, contribui para revelar o erro de um programa Prolog. A Seção 5 contém as conclusões e contribuições do trabalho.

2 O Critério Análise de Mutantes

A Técnica baseada em erros procura utilizar-se de certos tipos de erros que ocorrem durante o processo de desenvolvimento de um software para derivar casos de teste e assim verificar a presença ou ausência desses erros. Das diversas técnicas existentes, o critério *Análise de Mutantes* é um dos mais promissores para a atividade de teste. Este critério têm origem na década de 70 e possui semelhanças com um método clássico para detectar erros em circuitos digitais [VBD+97].

Em seu artigo publicado em 1978, DeMillo [DLS78] estabeleceu os princípios básicos do critério Análise de Mutantes. Para isto, utilizou-se de dois pressupostos para fundamentar o seu critério, são eles: **hipótese do programador competente** – "programadores experientes escrevem programas corretos ou muito próximos do correto". Desta forma, erros são propositadamente introduzidos em um programa de maneira a provocar uma alteração sintática válida[2]; e **efeito de acoplamento** – "erros complexos são decorrentes de erros simples". Assim, espera-se que se existe um conjunto de testes que detecte erros simples em um programa então este poderá também revelar erros complexos.

A idéia básica do critério Análise de Mutantes é gerar para um programa P em teste vários programas $P_1, P_2, \ldots, P_n$ que diferem de P por pequenas alterações sintáticas (simulando erros simples), cada programa P_i é dito *mutante* de P. Para satisfazer o critério é necessário gerar um dado de teste t que faça com que cada mutante comporte-se diferentemente de P. Neste caso, cada programa mutante P_i é dito *morto*. Quando não existir nenhum dado de teste capaz de distingüir um mutante P_i do programa em teste P, dizemos que o mutante P_i é *equivalente* à P. Aliado ao alto custo computacional e espacial, a existência de mutantes equivalentes dificulta a utilização prática do critério. Algumas heurísticas foram propostas para contornar este problema em uma grande porcentagem dos casos [OC94, OP96].

O critério Análise de Mutante, além de auxiliar no desenvolvimento de um conjunto de dados de teste T, pode ser utilizado para avaliar a adequação deste conjunto. Através de um *escore de mutação*, pode-se obter uma medida objetiva do nível de confiaça que podemos ter em T. O score é dado em termos do número de mutantes mortos e do número de mutantes gerados a partir de P. Deve-se ressaltar que a existência de mutantes equivalentes diminui consideravelmente a cobertura do critério, fazendo com que o escore de mutação seja sempre menor que a unidade.

[1] Inductive Logic Programming

[2] Ou seja, que não gere um erro de sintaxe

Quando consideram-se métodos práticos para a geração de mutantes, utiliza-se o conceito de operadores de mutação. Um *operador de mutação* é uma transformação sintática aplicada a uma determinada estrutura em um programa P, trocando o seu conteúdo.

As primeiras ferramentas automatizadas que aplicaram o critério Análise de Mutantes surgiram a partir de 1979. Tais ferramentas apoiavam o teste de programas desenvolvidos em Fortran e Cobol. A partir dos anos 80, ferramentas mais avançadas foram desenvolvidas. Por exemplo, a ferramenta *Mothra* é uma ferramenta de teste para programas escritos em Fortran desenvolvido pela *Purdue University* e o *Georgia Institute of Technology* [DMGK88]. A ferramenta *Proteum (Program Testing Using Mutantes)* é uma ferramenta de teste para programas escritos na linguagem C desenvolvida pelo *Grupo de Testes do Instituto de Ciências Matemáticas de São Carlos* [Del93]. Vale ressaltar que essas ferramentas utilizam operadores de mutação para linguagens procedurais e nao permitem o teste de programas Prolog.

3 Proposta de Operadores de Mutação para a Linguagem Prolog

Nesta Seção, é apresentado um conjunto de operadores de mutação para a linguagem Prolog. Esse conjunto, é resultado de estudos realizados e não tem o objetivo de ser completo, nem ser o melhor, experimentos deverão ser realizados em trabalhos futuros com o objetivo de aprimoramento e avaliação do mesmo.

É importante observar que, a cada operador de mutação, está associado um parâmetro numérico que descreve a porcentagem de aplicação do operador. Esta aplicação corresponde as possíveis combinações que o operador poderá gerar em um procedimento Prolog[3]

Os operadores propostos foram divididos em 4 grupos, bascando-se fortemente nos operadores da Ferramenta Proteum: mutação em cláusulas, em operadores, em variáveis e constantes.

3.1 Mutação em Cláusulas

PD: remove um predicado P em uma cláusula C.

Entrada

```
writel([]).
writel([H|T]) :- write(H),nl,
  writel(T).
```

Saída

```
writel([]).
writel([H|T]) :- nl,
  writel(T).
```

SP: troca a ordem de predicados adjacentes P_1 e P_2 em uma cláusula C.

Entrada

```
likes(ana,X) :- toy(X),plays(ana,X).
```

Saída

```
likes(ana,X) :- plays(ana,X),toy(X).
```

CDR: troca a conjunção entre predicados P_1 e P_2 por uma disjunção em uma cláusula C.

Entrada

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

Saída

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

DCR: troca a disjunção entre predicados P_1 e P_2 por uma conjunção em uma cláusula C.

Entrada

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

Saída

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

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

[3] Um procedimento em Prolog corresponde a um conjunto de fatos ou regras (cláusulas) cujos predicados que antecedem o operador : – possuem o mesmo nome.

Entrada

```
bubble_sort(L,L1) :- swap(L,L2,0),
  bubble_sort(L2,L1).
bubble_sort(L,L).
```

Saída

```
bubble_sort(L,L1) :- swap(L,L2,0),!,
  bubble_sort(L2,L1).
bubble_sort(L,L).
```

RC: remove uma ocorrência do "cut" (!) em uma cláusula C.

Entrada

```
not(G) :- G,!,fail.
not(G).
```

Saída

```
not(G) :- G,fail.
not(G).
```

PC: permuta uma ocorrência do "cut" (!) em uma cláusula C.

Entrada

```
insert(X,[H|T],[H|T1] :- !,X>H,
  insert(X,T,T1).
```

Saída

```
insert(X,[H|T],[H|T1] :- X>H,!,
  insert(X,T,T1).
```

3.2 Mutação em Operadores

AOM: troca um operador aritmético por outro operador aritmético em uma cláusula C.

Entrada

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

Saída

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

ROM: troca um operador relacional por outro operador relacional em uma cláusula C.

Entrada

```
fault(X) :- non(respond(X,Y)),X\==Y.
```

Saída

```
fault(X) :- non(respond(X,Y)),X>Y.
```

3.3 Mutação em Variáveis

VV: troca uma variável em um predicado P por uma outra variável em uma cláusula C.

Entrada

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

Saída

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

VVA: troca uma variável em um predicado P por uma variável anônima em uma cláusula C.

Entrada

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

Saída

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

VAV: troca uma variável anônima em um predicado P por uma variável normal em C.

Entrada

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

Saída

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

3.4 Mutação em Constantes

CCR: troca uma constante em um predicado P por uma outra constante em uma cláusula C.

Entrada

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

Saída

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

CVA: troca uma constante em um predicado P por uma variável anônima em uma cláusula C.

Entrada

```
likes(ana,jose). likes(jose,ana).
```

Saída

```
likes(ana,jose). likes(jose,_).
```

VAC: troca uma variável anônima em um predicado P por uma constante em uma cláusula C.

Entrada

```
likes(ana,apple). likes(paulo,_).
```

Saída

```
likes(ana,apple). likes(paulo,apple).
```

CV: troca uma constante em um predicado P por uma variável em uma cláusula C.

Entrada

```
append([],L,L).
append([H|T1],L2,[H|T]) :-
  append(T1,L2,T).
```

Saída

```
append(L2,L,L).
append([H|T1],L2,[H|T]) :-
  append(T1,L2,T).
```

VC: troca uma variável em um predicado P por uma constante em uma cláusula C.

Entrada

```
nth(N,X,[X|T]).
nth(N,X,[Y|T]) :- nth(M,X,T),N is M+1.
```

Saída

```
nth(1,X,[X|T]).
nth(N,X,[Y|T]) :- nth(M,X,T),N is M+1.
```

4 Exemplo de Aplicação

Esta seção mostra um exemplo de um programa Prolog para o qual a aplicação do critério Análise de Mutantes, auxilia a revelar um erro. A Figura 1 apresenta o procedimento incorreto *qsort(L1,L2)* que dada uma lista *L1* de entrada, o algoritmo devolve em *L2* a lista *L1* ordenada. O programa apresenta um erro de utilização do operador relacional; a forma correta é apresentada entre comentários.

Figura 1: **Programa Prolog para ordenar uma lista**

```
qsort(L,L1) :- quick_sort(L,[],L1).

quick_sort([],R,R).
quick_sort([H|T],R,L1) :-
  partition(T,Lt,H,Gt),
  quick_sort(Gt,R,GtsR),
  quick_sort(Lt,[H|GtsR],L1).

partition([],[],_,[]).
partition([H|T],[H|TLt],X,Gt) :-
  H < X, partition(T,TLt,X,Gt).  /* forma correta: H =< X */
partition([H|T],Lt,X,[H|TGt]) :-
  H > X, partition(T,Lt,X,TGt).
```

O mutante que difere do programa em teste trocando < por <= descreve o erro e está entre os mutantes gerados pelos operadores definidos na seção anterior. O dado de teste [1,3,2,1] que mata esse mutante produzirá a saída incorreta *no* e revela o erro.

5 Conclusão

O teste de programas Prolog tradicionalmente não envolve a aplicação de um critério de teste e existem poucas ferramentas que auxiliam essa atividade. A técnica baseada em erros parece ser promissora, visto os resultados apresentados em [BG96a, BG96b] e pode sistematizar o teste desses programas ou outros programas que utilizam paradigmas similares.

A possibilidade de aplicação de um critéiro de teste, tal como a Análise de Mutantes é bastante importante pois permite que medidas de cobertura sejam fornecidas para avaliar a qualidade dos dados de teste utilizados. O primeiro passo para a aplicação do critério Análise de Mutantes é a definição de operadores de mutação que é o objetivo deste trabalho. Esse conjunto, é resultado de estudos realizados e não tem o objetivo de ser completo, nem ser o melhor.

O trabalho e os operadores propostos tornam possível a implementação de uma ferramenta de teste e que trabalhos futuros realizem experimentos para avaliar a eficácia do critério Análise de Mutantes no teste de programas Prolog e que também o próprio conjunto de operadores seja refinado. Os operadores poderão ser adaptados para teste de programas escritos em linguagens similares e cuja programação esteja sujeita aos mesmos tipos de erros que a programação Prolog. Isto poderá sistematizar o teste desses programas e facilitar a avaliação da adequação de um dado conjunto de testes.

Referências

[BG96a] F. Bergadano and D. Gunetti. *Inductive Logic Programming: From Machine Learning to Software Engineering.* The MIT Press, 1996.

[BG96b] F. Bergadano and D. Gunetti. Testing by means of inductive program. *ACM Transactions on Software Engineering and Methodology*, 5(2):119–145, April 1996.

[DDH72] O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare. Notes on structured programming. In *Structured Programming.* Academic Press, 1972.

[Del93] M.E. Delamaro. *Proteum: Um ambiente de teste baseado na Análise de Mutantes.* Master Thesis, ICMSC-USP-São Carlos, São Carlos - SP, Brazil, March 1993. (in Portuguese).

[DLS78] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, Vol. 11(4):34–41, April 1978.

[DMGK88] R.A. De Millo, D.C. Gwind, and K.N. King. An extented overview of the mothra software testing environment. In *Proc. of the Second Workshop on Software Testing, Verification and Analysis*, pages 142–151. Computer Science Press, Banff - Canada, July 19-21 1988.

[Mal91] J.C. Maldonado. *Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software.* Doctorate Dissertation, DCA/FEEC/Unicamp, Campinas - SP, Brazil, July 1991. (in Portuguese).

[McC76] T. McCabe. A software complexity measure. *IEEE Transactions on Software Engineering*, Vol. SE-2(4):308–320, December 1976.

[MdR94] S. Muggleton and L. de Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, pages 629–679, 1994.

[Mye79] G.J. Myers. *The Art of Software Testing.* Wiley, New York, 1979.

[OC94] A.J. Offutt and W.M. Craft. Using compiler optimization techniques to detect equivalent mutantes. *The Journal of Software Testing, Verification, and Reliability*, 4(3):131–154, September 1994.

[OP96] A.J. Offutt and J. Pan. Detecting equivalent mutants and the feasible path problem. In IEEE Computer Society Press, editor, *Proceedings of the 1996 Annual Conference on Computer Assurance (COMPASS 96)*, pages 224–236. UK, June 1996.

[Pre97] R. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, USA, fourth edition, 1997.

[RW85] S. Rapps and E.J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.

[TLK92] R.N. Taylor, D.L. Levine, and C.D. Kelly. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering*, Vol. SE-2(18(3)):206–215, 1992.

[VBD+97] A.M.R. Vincenzi, E.F. Barbosa, M.E. Delamaro, S.R.C de Souza, and J.C. Maldonado. Critrio anlise de mutantes: Estado atual e perspectivas. In *Workshop do Projeto Validao e Teste de Sistemas de Operao*, pages 15–26. guas de Lindia, Janeiro 1997.

[Ver97] S.R. Vergilio. *Critérios Restritos de Teste de Software: Uma Contribuição para Gerar Dados de Teste mais Eficazes.* Doctorate Dissertation, DCA/FEEC/Unicamp, Campinas - SP, Brazil, July 1997. (in Portuguese).