Gerando Dados de Teste para Programas Orientados a Objeto com Um Algoritmo Genético Multi-Objetivo

Gustavo Henrique de Lima Pinto[1], **Silvia Regina Vergilio**[1]

[1]Departamento de Informática – Universidade Federal do Paraná (UFPR)
Caixa Postal 19.081 – 81.531-980 – Curitiba – PR – Brasil

{gustavo,silvia}@inf.ufpr.br

Resumo. *O teste evolutivo de software usa algoritmos de busca heurística para a geração de dados de teste. Os diversos trabalhos existentes utilizam diferentes técnicas e funções de fitness. Entretanto, as funções utilizadas são baseadas em um único objetivo, geralmente a cobertura de um critério de teste. Mas existem diferentes fatores que podem influenciar a tarefa de geração de dados de teste. Para considerar estes fatores, este trabalho descreve um algoritmo genético multiobjetivo, integrado com a ferramenta de teste JABUTi que apóia o teste estrutural de programas orientados a objeto. Dois objetivos são avaliados experimentalmente, a cobertura de um critério de teste e o tempo de execução. Os resultados mostram um bom desempenho do algoritmo implementado.*

1. Introdução

A tarefa de geração de dados de teste para satisfazer um dado critério de teste geralmente consome muito esforço e não pode ser completamente automatizada. Isso se deve a inúmeras limitações da atividade de teste, tais como a existência de caminhos não executáveis. Devido a isto, o tema geração de dados de teste, ainda é um desafio, associado a inúmeros trabalhos científicos. Dentre estes, trabalhos que utilizam algoritmos de busca, tais como Algoritmos Genéticos e outros evolutivos, têm apresentado bastante sucesso, o que originou uma nova área de pesquisa chamada Teste Evolutivo de software [McMinn 2004]. Os trabalhos destas áreas diferem no tipo de técnica utilizada, e na função de fitness utilizada. A maioria dos trabalhos são orientados a cobertura de critérios de teste estruturais no teste de programas procedurais [McMinn 2004]. No contexto de programas orientados a objeto, existe um número mais reduzido de trabalhos, destacando-se [Araki and Vergilio 2010, Ribeiro 2008, Sagarna et al. 2007, Seesing and Gross 2006, Tonella 2004, Wappler and Wegener 2006].

O problema desses trabalhos é que a maioria deles não está integrado a uma ferramenta de teste. Além disso, todos eles tratam a geração de dados de teste como um problema de um único objetivo, geralmente a cobertura de um critério, e utilizam uma única função de fitness. Entretanto, existem diversos fatores que precisam ser considerados na geração de dados de teste, tais como a capacidade do dado de teste revelar certos tipos de defeitos, o tempo de execução, consumo de memória, e outros que podem ser específicos do tipo de programa sendo desenvolvido e ambiente de desenvolvimento.

Portanto percebe-se que o problema de geração de dados de teste é um problema que envolve múltiplas variáveis (objetivos) e para o qual uma solução ótima satisfazendo todos os objetivos nem sempre é possível, pois eles podem ser conflitantes. Para avaliar

e determinar o grupo destas boas soluções, utiliza-se o conceito de dominância de Pareto [Pareto 1927]. A idéia é descobrir soluções que não são dominadas por nenhuma outra no espaço de busca. Dado um conjunto de possíveis soluções para um problema, uma solução A domina uma solução B, se e somente se, A é melhor que B em satisfazer pelo menos um dos objetivos, sem ser pior que B em nenhum dos outros objetivos.

Na Engenharia de Software a utilização de algoritmos de otimização multiobjetivo é um tema emergente de pesquisa. No teste de software, eles têm sido utilizados para priorização de casos de teste [Yoo and Harman 2007] e também para geração de dados de teste [Cao et al. 2009, Ghiduk et al. 2007, Harman et al. 2007]. Esses trabalhos apresentam resultados promissores, entretanto, eles não se encontram integrados a uma ferramenta de teste, nem consideram o contexto de programas orientados a objetos ou a cobertura de diferentes critérios de teste, e além disto, há outros fatores não investigados a serem considerados na geração de dados de teste. Diferentemente dos trabalhos encontrados na literatura, o presente trabalho apresenta uma proposta de geração de dados de teste com algoritmos multiobjetivos no contexto de programas orientados a objeto, descrevendo um framework integrado a uma ferramenta de teste, utilizando como objetivos, a cobertura e tempo de execução. O algoritmo implementado é o NSGA-II (Nondominated Sorting Genetic Algorithm) concebido como uma modificação no NSGA proposto em [Deb and Srinivas 1994]. Resultados experimentais mostram que o algoritmo implementado é melhor que uma geração aleatória e que consegue apresentar um conjunto de boas soluções para ambos objetivos.

O trabalho está organizado como segue. Na Seção 2 alguns detalhes de implementação do algoritmo são descritos. Na Seção 3 são apresentados os resultados experimentais. Na Seção 4 estão as conclusões e possíveis trabalhos futuros.

2. A Geração de Dados de Teste Como Um Problema Multiobjetivo

Diversos fatores devem ser considerados para selecionar um dado de teste, por exemplo, é desejado maximizar a cobertura de um critério de teste e a capacidade do teste em revelar defeitos. Deve-se também procurar minimizar o tempo de execução, ou o tamanho do dado de teste, ou ainda o conjunto de dados de teste.

Esta seção descreve um algoritmo integrado com a ferramenta de teste JaBUTi (Java Bytecode Understanding and Testing) [Vincenzi et al. 2006] que apoia o teste estrutural em programas Java. O algoritmo implementa o NSGA-II, que é um algoritmo genético multiobjetivo dos mais conhecidos e utilizados e considera dois objetivos conflitantes: maximizar a cobertura de um critério de teste implementado pela JaBUTi e minimizar o tempo de execução do dado de teste.

Em geral o indivíduo na população representa um dado de teste. Neste trabalho, foi adotado um formato diferenciado para o indivíduo, que será um conjunto de dados de teste com um tamanho definido pelo usuário. Com essa abordagem procura-se aumentar as chances de encontrar boas soluções no espaço de busca. A codificação do dado de teste para programas orientados a objetos é uma tarefa complexa, pois não trata somente de uma seqüência de valores de entrada, como o teste de programas procedurais. O dado de teste deve ser codificado de forma que possa ser decodificado posteriormente, sem perder informações sobre construtores, invocações de métodos e valores passados por parâmetro. A representação escolhida para cada elemento do indivíduo é exemplificada na gramática

ilustrada a seguir, baseada em [Tonella 2004].

```
<chromosome>  ::=  <actions> @ <values>
<actions>     ::=  <action> {: <actions>}?
<action>      ::=  $id = constructor ( {<parameters>}? )
                |  $id = class # null
                |  $id . method ( {<parameters>? )
<parameters>  ::=  builtin-type {<generator>}?
                |  $id
<generator>   ::=  [ low ; up ]
                |  [ genClass ]
<values>      ::=  <value> {, <values>}?
<value>       ::=  integer
                |  real
                |  boolean
                |  string
```

Para exemplificação, seque um dado de teste resultante da codificação para o programa TriTyp.java. Este programa possui como entrada três números inteiros positivos e verifica se estes formam um triângulo equilátero, escaleno, isósceles ou se não formam um triângulo. Os métodos que atribuem os valores para os lados do triângulo são setI, setJ, setK, cujas invocações de métodos são separados por ':'. Após o @, são passados os valores referentes aos métodos, estes limitados aos tipos primitivos da linguagem Java.

```
$t=TriTyp() : $t.setI(int) : $t.setJ(int) : $t.setK(int) : $t.type() @ 9, 3, 8
```

As seguintes funções de fitness foram utilizadas: a) Tempo de execução: refere-se à minimização do tempo de execução de um dado de teste, que é o período em que o teste permanece em execução (tempo final - tempo inicial); b) Cobertura de um critério: A cobertura de um critério é mensurada com auxílio da ferramenta de teste JaBUTi e tem como foco a maximização. O cálculo da cobertura do critério é feito pela fórmula:

$$Fitness_x = \frac{\text{nr. elementos cobertos}_x \text{ * 100}}{\text{nr. total de elementos requeridos}}$$

Inicialmente, o usuário testador deverá fornecer os parâmetros do algoritmo. Tais parâmetros são separados em dois conjuntos. O primeiro conjunto é referente aos parâmetros do algoritmo evolutivo, e incluem: número de gerações, número de dados de teste que irão compor o indivíduo, objetivos e probabilidades relacionadas aos operadores genéticos. Os operadores de mutação realizam mudanças (ou remoções) nos valores de entrada, em chamadas de construtores e invocações de métodos. O operador de crossover de ponto único foi implementado. Esses operadores estão descritos em [Araki and Vergilio 2010, Tonella 2004]. Foi ainda implementado um operador especial, chamado de Relacionamento. Sua finalidade é alternar dados de teste entre indivíduos para garantir a diversidade da população como um todo. Este conta ainda com dois métodos: Half relation e Random relation. Para ilustração, considere como exemplo inicial o conjunto A = {a, b, c, d, e, f} e conjunto B = {g, h, i, j, k, l}. De acordo com a abordagem *half relation* os novos conjuntos derivados dessa união deverão conter a primeira metade do conjunto A, e a segunda metade do conjunto B. Como resultado, haverá os conjuntos HR_i = {a, b, c, j, k, l} e HR_{ii} = {g, h, i, d, e, f}. Por outro lado, se adotada a estratégia *Random relation*, serão sorteados os índices dos elementos que deverão ser alternados.

O segundo conjunto de parâmetros diz respeito aos parâmetros de teste: unidade (classe) a ser testada (cut) e um dos critérios de teste da ferramenta Ja-

BUTi [Vincenzi et al. 2006] que são: todos-nos, todos-arcos, todos-usos e todos-potenciais-usos.

Dados de testes são gerados e codificados de maneira que um único indivíduo represente um conjunto de dados de teste. Posteriormente, as principais variáveis são inicializadas e é criada a população inicial aleatoriamente. Um grande laço dá início ao processo de evolução. No primeiro momento, cada elemento (dado de teste) do indivíduo deve ser avaliado pela ferramenta individualmente afim de encontrar o fitness de cada elemento. Após, é avaliado o fitness do indivíduo, que nada mais é que a relação entre os fitness de todos os dados de teste. Posteriormente são identificadas as soluções dominadas e não-dominadas, de acordo com o critério de dominância de Pareto. Após a identificação, é aplicado o critério de seleção multiobjetivo e os individuos selecionados darão início à outra evolução, a dos dados de teste.

Neste momento, aplica-se a técnica da roleta para selecionar os dados de teste do indivíduo que sofrerão a ação dos operadores genéticos (mutação e cruzamento). Inicia-se então uma evolução referente ao conjunto de dados de teste, pois espera-se que o conjunto retenha os melhores testes no decorrer das iterações. No final da iteração, os dados de teste são alternados entre os indivíduos através da técnica denominada de Relacionamento, preservando assim a diversidade. Essa etapa caracteriza a segunda evolução do algoritmo, relacionada ao aperfeiçoamento dos conjuntos dos dados de teste.

Finalmente o grupo das melhores soluções são acrescentadas em uma nova população, e por conseguinte, apresentadas ao usuário. O ciclo será retomado até que o critério de parada seja satisfeito, ou seja, um número máximo de gerações for atingido. Após o fim do laço, uma última avaliação é feita nos indivíduos da última geração, e as soluções não dominadas por hora encontradas, são adicionadas novamente no montante de soluções não dominadas encontradas durante todo o processo.

3. Experimentos Realizados

Para avaliar o algoritmo implementado, alguns experimentos foram conduzidos com dois programas Java: TriTyp.java e Find.java (que busca por um inteiro em um vetor). Os programas foram extraídos do site http://www.inf-cr.uclm.es/www/mpolo/stvr/ e têm sido utilizados em vários trabalhos da literatura [Polo et al. 2009] e apesar de simples dão uma idéia do funcionamento do algoritmo. Nestes experimentos procurou-se avaliar dois tipos de critérios: o critério baseado em fluxo de controle todos-arcos (AC) e o critério baseado em fluxo de dados todos-usos (US).

A configuração foi ajustada empiricamente (Tabela 1). Os resultados obtidos pelo NSGA-II foram comparados com uma estratégia aleatória configurando o número de gerações para 1, visto que a primeira população é obtida aleatoriamente. Ambas estratégias foram executadas 10 vezes e após isto, o conjunto de soluções não dominadas foi obtido considerando todas as execuções. As soluções encontradas para cada programa e critério estão na Tabela 2.

Para o programa TriTyp e o critério todos-arcos o NSGA-II apresentou 5 soluções não dominadas e a estratégia aleatória apenas uma, sendo esta dominada pela solução do algoritmo genético. As soluções do algoritmo genético variam de 278 a 344ms com cobertura de 52 a 81%. O NSGA-II selecionou os pontos mais relevantes no espaço de

Tabela 1. Configurações do Algoritmo

Programa	Critério	T. Cross	T. Muta	T. Relation	numGerações	numIndividuos	numDadosTeste
TriTyp	AC	0.8	0.2	0.7	100	100	50
TriTyp	US	0.8	0.2	0.7	100	100	50
Find	AC	0.8	0.2	0.7	50	50	50
Find	US	0.8	0.2	0.7	50	100	50

Tabela 2. Resultados

Programa	Critério	NSGA-II		RS	
		Cov.(%)	Exec.Time (ms)	Cov.(%)	Exec.Time (ms)
TriTyp	AC	81	319	65	284
		76	297		
		74	281		
		72	278		
		52	344		
	US	69	315	42	349
				40	331
Find	AC	95	297	88	327
		93	292		
		88	303	86	247
	US	87	220	76	240
		77	200		

busca. Para o critério todos-usos, o NSGA-II apresentou somente uma solução que tambem domina ambas soluções encontradas pela estratégia aleatória. Resultados similares foram obtidos para o programa Find. NSGA-II apresentou uma maior número de soluções que domina todas a soluções encontradas pela estratégia aleatória. Observa-se como esperado que é mais dífícil satisfazer o critério baseado em fluxo de dados e que o NSGA melhorou em torno de 20% a cobertura obtida pela estratégia aleatória.

4. Conclusões

Este trabalho explorou uma abordagem que utiliza algoritmos genéticos multi-objetivos para geração de dados de teste no contexto de programas orientados a objetos. Foi implementado o algoritmo NSGA-II e dois objetivos: cobertura e tempo de execução no teste de programas Java. O algoritmo foi integrado a uma ferramenta de teste, que extrai informações de teste diretamente do código objeto Java, o que permite o teste mesmo na ausência do código fonte, o que é muito comum no teste de componentes. A representação do indivíduo também é independente do código fonte, as funções objetivo implementadas também permitem o uso dos diferentes critérios implementados pela JaBUTi.

Resultados experimentais foram conduzidos com os dois objetivos implementados. Os resultados mostram que as soluções obtidas pelo algoritmo NSGA-II são melhores do que as soluções obtidas por uma estratégia aleatória. Além disso, um número maior de soluções foi obtido, sendo estas bem distribuídas no espaço de busca, o que dá ao testador diferentes opções (soluções) não dominadas para escolher.

O algoritmo implementado poderá ser especializado e aperfeiçoado para tratar de outros problemas que envolvam a geração de dados de teste e/ou otimização multiobjetivo, podendo este ser aprofundado, aumentando o número de objetivos ou diversificando-os, como tratando outras questões como consumo de memória, ou teste baseado em defeitos. Além disso, novos experimentos com programas mais complexos e outros contextos

deverão ser conduzidos.

Referências

Araki, L. and Vergilio, S. R. (2010). Um framework de geração de dados de teste para critérios estruturais em código objeto Java. In *Workshop de Testes e Tolerância a Falhas*.

Cao, Y., Hu, C., and Li, L. (2009). Search-based multi-paths test data generation for structure-oriented testing. In *ACM/SIGEVO Summit on Genetic and Evolutionary Computation*, pages 25–32.

Deb, K. and Srinivas, N. (1994). Multiobjective optimization using nondominated sorting in genetic algorithms. In *IEEE Transactions on Evolutionary Computation*, pages 221–248.

Ghiduk, A. S., Harrold, M. J., and Girgis, M. R. (2007). Using genetic algorithms to aid test-data generation for data-flow coverage. In *Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC '07)*, pages 41–48.

Harman, M., Lakhotia, K., and McMinn, P. (2007). A multi-objective approach to search-based test data generation. In *Genetic and Evolutionary Computation Conference*.

McMinn, P. (2004). Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 2(14):105–156.

Pareto, V., editor (1927). *Manuel D"Economie Politique*. Ams Pr.

Polo, M., Piattini, M., and García-Rodríguez, I. (2009). Decreasing the cost of mutation testing with second-order mutants. *Software Testing Verification Reliability*, 19(2):111–131.

Ribeiro, J. C. B. (2008). Search-based test case generation for object-oriented Java software using strongly-typed genetic programming. In *Genetic and Evolutionary Computation Conference*.

Sagarna, R., Arcuri, A., and Yao, X. (2007). Estimation of distribution algorithms for testing object oriented software. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC '07)*, pages 438–444.

Seesing, A. and Gross, H.-G. (2006). A genetic programming approach to automated test generation for object-oriented software. *International Transactions on System Science and Applications*, 1(2):127–134.

Tonella, P. (2004). Evolutionary testing of classes. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 119–128.

Vincenzi, A. M. R., Delamaro, M. E., Maldonado, J. C., and Wong, W. E. (2006). Establishing structural testing criteria for Java bytecode. *Software: Practice and Experience*, 36(14):1513–1541.

Wappler, S. and Wegener, J. (2006). Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In *Genetic and Evolutionary Computation Conference*.

Yoo, S. and Harman, M. (2007). Pareto efficient multi-objective test case selection. In *International Symposium on Software Testing and Analysis*.