# Um Framework de Geração de Dados de Teste para Critérios Estruturais Baseados em Código Objeto Java

**Lucilia Yoshie Araki[1], Silvia Regina Vergilio[1]**

[1]Departamento de Informática – Universidade Federal do Paraná (UFPR)
Caixa Postal 19.081 – 81531 - 970 – Curitiba – PR

{lya_araki@yahoo.com.br, silvia@inf.ufpr.br}

***Resumo.*** *O teste evolutivo de software orientado a objeto é uma área de pesquisa emergente. Algumas abordagens promissoras sobre o assunto são encontradas na literatura, entretanto, estas não consideram critérios propostos recentemente que utilizam o código objeto Java para obter os requisitos de teste. Além disso, os trabalhos geralmente não estão integrados a uma ferramenta de teste. Neste artigo, um framework, chamado TDSGen/OO para geração de dados de teste é descrito. TDSGen/OO utiliza Algoritmos Genéticos e trabalha de maneira integrada com a ferramenta JaBUTi, que implementa diferentes critérios de teste baseados no bytecode e em mecanismos de tratamento de exceções, permitindo o teste de componentes mesmo que o código fonte não esteja disponível. Alguns resultados preliminares são também apresentados que mostram benefícios no uso do framework.*

***Abstract.*** *The evolutionary test of object-oriented software is an emergent research area. We find in the literature promising approaches on this subject, however, those approaches do not consider some recent test criteria that use the Java Byte-code to derive the test requirements. In addition to this, they are not usually integrated to a test tool. In this paper, we describe a framework, named TDSGen/OO to test data generation. The framework uses a Genetic Algorithm and is integrated with JaBUTi, a tool that implements different test criteria based on bytecode and exception-handling mechanisms, allowing the test of components even if the source code is not available. Some preliminary evaluation results are also presented.*

## 1. Introdução

Nos últimos anos, o paradigma de orientação a objeto ganhou importância e vem sendo intensivamente utilizado. O uso de recursos específicos deste paradigma pode introduzir novos e diferentes tipos de defeitos. Por isso, a aplicação de um critério de teste neste contexto é fundamental.

Diferentes critérios foram propostos. Eles utilizam diferentes tipos de informação para derivar os requisitos de teste. Muitos critérios são baseados na especificação [13] e em diversos modelos, como diagrama de estados, casos de uso, classes, e etc. Outros critérios são baseados no programa. Os critérios estruturais são

geralmente baseados nos grafos de fluxo de controle e interações de fluxo de dados, considerando sequências de chamadas de métodos [6].

Recentemente, foram propostos alguns critérios estruturais de teste que consideram informações do código objeto Java [21]. Estes critérios podem ser aplicados mesmo quando o código fonte não está disponível. Isso é bastante comum na maioria dos testes de componentes de software, o que torna um critério baseado em bytecode muito útil. Além disso, uma ferramenta, chamada JaBUTi (Java Bytecode Understanding and Testing) [20, 21], está disponível para aplicação de tais critérios. JaBUTi implementa critérios baseados em fluxo de controle e em fluxo de dados considerando mecanismo de manipulação de exceções.

A ferramenta tem como objetivo o teste de unidade de uma determinada classe. Gera os elementos requeridos pelos critérios de aplicação, mostra o gráfico correspondente, e produz relatórios de cobertura com respeito ao conjunto de teste fornecido pelo testador, que é responsável pela geração manual dos dados de teste para cobrir um critério de teste específico. Esta tarefa torna o teste demorado, difícil e caro e, em geral, é feita manualmente.

Esta limitação é tema de pesquisa da área denominada Teste Evolucionário [25], que aplica algoritmos evolutivos para geração de dados de teste. A maioria dos trabalhos em teste evolucionário, no entanto, destinam-se ao teste de unidade do código procedural [1,9,10,11]. Podemos citar, entre estes trabalhos, a ferramenta TDSGen [3], que gera dados de teste para critérios estruturais e critérios baseados em defeito, para programas em C. TSDGen implementa mecanismos de hibridização para melhorar o desempenho do algoritmo genético, e trabalha com duas ferramentas de teste.

No contexto de software orientado a objeto, o teste evolucionário é considerado uma área emergente de pesquisa e ainda não foi investigado adequadamente [15]. Há um número reduzido de trabalhos. Tonella [19] foi o primeiro a investigar o teste evolutivo de classes usando Algoritmos Genéticos (GA) para satisfazer o critério estrutural todos-os-ramos. Outros trabalhos investigam outras técnicas como: Colônia de Formigas [8], Algoritmos Evolutivos Universais [22], Programação Genética [18.24], Algoritmos de Distribuição de Estimativas [17], e etc. [23]. Estes trabalhos usam diferentes técnicas para diferentes finalidades. A maioria deles destina-se apenas a um critério de teste, geralmente o critério todos-nós ou todos-ramos. Eles não consideram os critérios baseados em bytecode, e também não oferecem uma implementação integrada com uma ferramenta que apóie o critério escolhido.

Devido a isso, neste trabalho, um framework para geração de dados de teste é apresentado. O framework, chamado TDSGen/OO (Test Data Set Generator for OO Software), tem como finalidade a geração de dados de teste para satisfazer os critérios baseados em fluxo de controle e em fluxo de dados que consideram o bytecode, implementados pela ferramenta de teste JaBUTi. O framework implementa um algoritmo genético e alguns mecanismos para melhorar o desempenho que foram utilizados com sucesso por TDSGEN [3] no teste de software procedural.

A idéia é fornecer um ambiente para a aplicação de todos os critérios executados em uma estratégia de teste, e reduzir o esforço e o custo do teste no contexto de software orientado ao objeto.

As demais seções deste artigo estão organizadas da seguinte forma. A Seção 2 discute trabalhos relacionados. Seção 3 descreve o framework TDSGen/OO. Resultados de um estudo preliminar são apresentados na Seção 4. As conclusões e trabalhos futuros estão na Seção 5.

## 2. Trabalhos Relacionados

Existem vários trabalhos sobre teste evolucionário de programas procedurais que podem ser encontrados na literatura, [1, 5, 9, 10]. As principais técnicas utilizadas por estes trabalhos para a geração de dados de teste são: Hill Climbing, Simulated Annealing, Algoritmos Genéticos, Programação Genética, e etc. Além das técnicas, outra diferença está na função de aptidão (ou fitness) usada. Alguns trabalhos usam uma função de fitness orientada à cobertura de um critério. Nesse caso, são abordados diferentes critérios, com base no: fluxo de controle e de dados, ou defeitos e teste de mutação. Outras funções têm o objetivo de cobrir um elemento específico exigido.

Por outro lado, no contexto de orientação a objeto, um número menor de trabalhos é encontrado. Poucas técnicas foram exploradas, bem como, poucos critérios e funções de fitness.

O primeiro trabalho a explorar o teste de software orientado a objeto foi a abordagem baseada em Algoritmos Genéticos, proposta por Tonella [19]. A abordagem propôs uma representação baseada em gramáticas (Figura 1) para os dados de teste, capaz de representar seqüência de invocações de métodos, além da simples representação das entradas utilizadas nos programas tradicionais.

```
<chromosome>   ::= <actions> @ <values>

<actions>   ::= <action> {: <actions>} ? <action>

::=  $id=constructor({<parameters>}? )

| $id = class # null

| $id . method( {<parameters>}? )

<parameters>::=  <parameter> {, <parameters>? }

<parameter>      ::=   builtin-type {<generator>} ? | $id

<generator>      ::=  [low ;  up] | [genClass]

<values>         ::= <value> {, <value>} ?

<value>          ::= integer | real |  boolean | string
```

**Figura 1: Gramática introduzida por Tonella (extraída de [19]).**

A gramática representa um dado de teste (ou cenário de teste) para software orientado a objeto. O "@" divide o cromossomo em duas partes. A primeira parte representa a seqüência de invocações de métodos ou construtores. A segunda parte contém os valores de entrada para essas operações, separadas por “,”. Uma ação pode representar um novo objeto $id  ou uma chamada para um método identificado por $id.

Os parâmetros podem representar tipos tais como inteiro, real, string ou booleanos. O trabalho também propôs um conjunto de operadores evolutivos de cruzamento (crossover) e mutação. A função de fitness é orientada a um objetivo.

O trabalho de Liu et al. [8] utiliza um algoritmo de Colônia de Formigas para gerar a menor seqüência de invocações de métodos. O objetivo é cobrir as arestas em métodos privados e protegidos.

Wappler e Lammermann [22] apresentam uma abordagem baseada em Algoritmos Evolutivos Universais para permitir a aplicação de diferentes algoritmos de busca, tais como Hill Climbing e Simulated Annealing. A função de fitness usa mecanismos para penalizar seqüências inválidas e guiar a busca, pois sua representação pode gerar indivíduos que poderiam ser transformados em programas incorretos. Um trabalho posterior de Wappler e Wegner [24] não apresenta esse problema e codifica possíveis soluções usando Programação Genética Fortemente Tipada. As seqüências são representadas por árvores, e resultados promissores foram obtidos. O uso de Programação Genética também foi explorado por Seesing e Gross [18].

Sagarna et al. [17] apresentam uma abordagem baseada em Algoritmos de Distribuição de Estimativas (EDA) para gerar dados de teste para a cobertura de um ramo no teste de unidade de programas Java.

Um problema com os trabalhos mencionados é que a representação dos dados de teste, assim como a abstração do fluxo de execução e/ou de dados estão baseadas no código fonte. Se o código não está disponível, os algoritmos não podem ser aplicados. Para superar esta limitação os trabalhos mais recentes tentam gerar dados de teste analisando diretamente o bytecode. O trabalho de Cheon et al. [2] propõe extensões ao compilador JML para produzir informações sobre cobertura e utiliza as especificações JML para determinar o resultado do teste. O trabalho descrito em [12] gera dados de teste resolvendo restrições. Estes trabalhos, entretanto, não usam algoritmos evolutivos, baseados em busca meta-heurística.

Os trabalhos relatados em [15,16] abordam o teste evolutivo considerando bytecode. Os trabalhos utilizam um pacote que inclui diferentes Algoritmos Evolutivos Universais e Programação Genética Fortemente Tipada, de forma semelhante ao trabalho descrito em [22]. A geração é guiada por informações extraídas do bytecode.

Na próxima seção, é descrito o framework TSDGen/OO para a geração de dados de teste no contexto de software orientado a objeto, integrado com a ferramenta de teste JaBUTi. A representação do indivíduo é baseada no trabalho de Tonella [19], mas diferentemente, a ferramenta executa uma função de fitness orientada à cobertura dos critérios estruturais baseados em fluxo de controle e em fluxo de dados, aplicados a um modelo extraído do bytecode. Um outro aspecto que torna o trabalho diferente dos demais trabalhos que também consideram o bytecode na geração [15.16] é a função implementada que permite uma melhoria no desempenho de um algoritmo genético simples, considerando trabalhos anteriores [3] e toda a experiência adquirida com programas tradicionais.

## 3. Ferramenta TDSGen/OO

TDSGen/OO gera dados de teste para os critérios estruturais implementados pela JaBUTi, que são critérios baseados em fluxo de controle e em fluxo de dados. A Figura 2 apresenta sua estrutura, baseada no trabalho relatado em [3]. TDSGen/OO contém quatro módulos principais: Geração da População, Avaliação, Evolução e Início. A informação produzida transmitida através dos módulos está representada nesta figura por elipses. Na sequência é apresentada uma breve descrição de cada módulo.

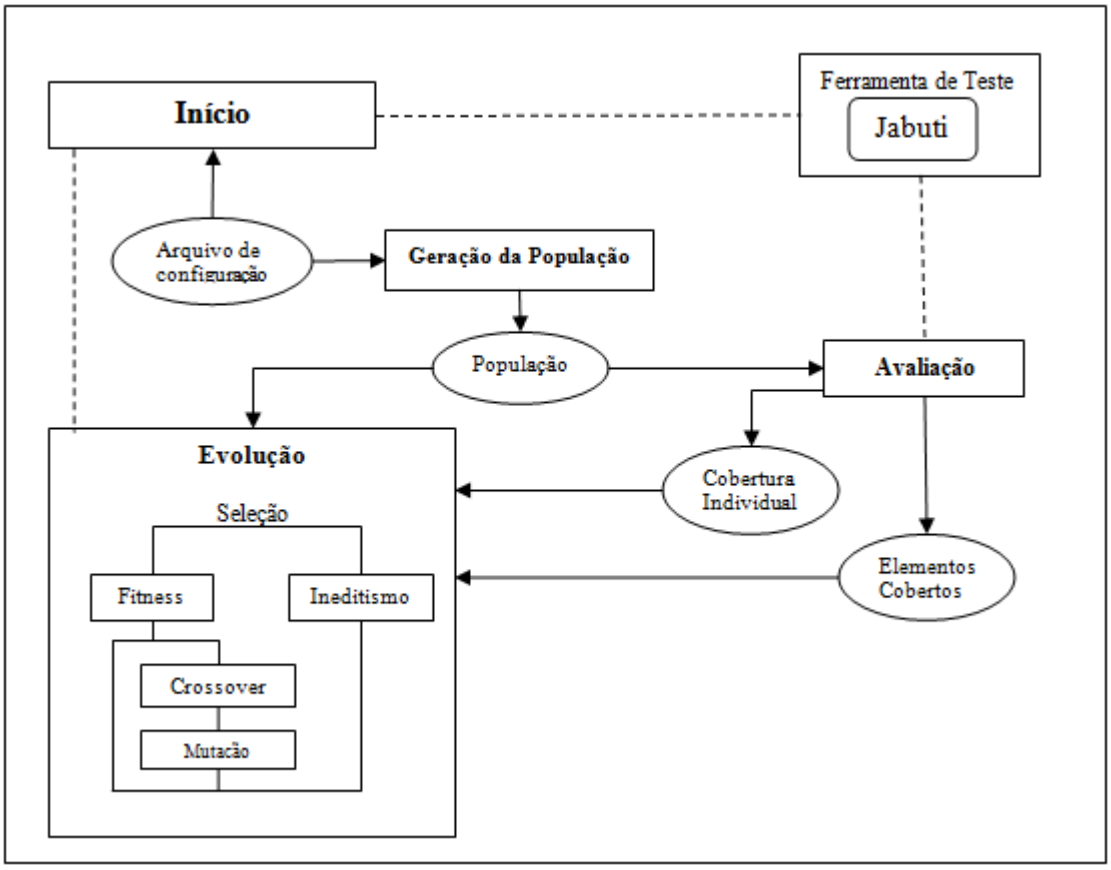


**Figura 2: Principais módulos do framework TDSGen/OO**

### A. Início

O Módulo Início é responsável por receber a configuração inicial (arquivo de configuração) do testador e por controlar os outros módulos. O testador pode também usar uma interface gráfica para fornecer as informações iniciais.

Uma configuração possui duas seções, sendo que uma seção - ferramenta de teste - inclui parâmetros para a JaBUTi: o nome do arquivo fonte que contém a classe a ser testada, e o critério escolhido; a outra seção - estratégias de evolução - está relacionada com o processo de evolução e será utilizada pelo Módulo Evolução, contendo taxa de crossover, taxa de mutação, tamanho da população, número máximo de gerações, método de seleção (torneio[1] ou roleta[2]), elitismo e ineditismo.

[1] Seleção por Torneio: um número p de indivíduos da população é escolhido aleatoriamente para formar uma sub-população temporária. Deste grupo, é selecionado o melhor indivíduo.

Um exemplo de configuração é apresentado na Figura 3. A população tem 50 indivíduos, ou seja, 50 candidados à solução do problema, com um tamanho máximo, relacionado ao número de invocações de métodos igual a 10. Elitismo e ineditismo não estão habilitados. As taxas de crossover são 0,75 e de mutação 0,01. No crossover dois indivíduos pais são selecionados e seu material genético é combinado, permutando uma parte de um dos pais por uma parte do outro, gerando um novo indivíduo. O operador de mutação modifica aleatoriamente um ou mais invocações de métodos do indivíduo.

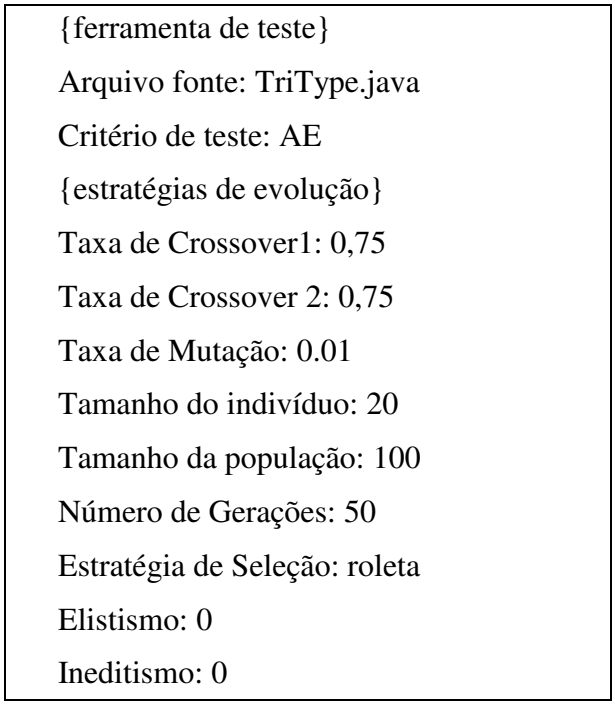


**Figura 3: Configuração básica usada pela TDSGen/OO**

## B. Geração da População

Este módulo trabalha de acordo com as informações contidas no arquivo de configuração. Ele gera a população inicial e também é responsável pela codificação e decodificação de cada indivíduo.

A representação escolhida para o cromossomo (indivíduo) usa a gramática introduzida por Tonella [19] apresentada na Seção 2. O módulo garante que os cromossomos sejam bem-formados, ou seja, contenham para cada chamada de método seu parâmetro correspondente; $id é sempre determinado antes da sua utilização, e sempre é associado a uma ação correspondente.

A população inicial é gerada aleatoriamente, sendo esta população um conjunto de possíveis dados de teste. A Figura 4 apresenta um exemplo da população com 3 indivíduos para o programa Trityp, que verifica se as suas três entradas compostas por números inteiros formam um triângulo. Em caso afirmativo, o programa imprime o tipo

---

[2] Seleção por Roleta: Especifica a probabilidade de que cada indivíduo seja selecionado para a próxima geração. Cada indivíduo da população recebe uma porção da roleta proporcional a sua probabilidade.

de triângulo formado. A primeira parte identificada por $ representa as chamadas de métodos do programa TriTyp e logo após o @ encontram-se os parâmetros necessários para cada chamada de método.

```
$t=TriTyp() : $t.setI(int)  : $t.setJ(int) : $t.setK(int) : $t.type() @ 5, 5, 5
$t=TriTyp() : $t.setI(int) : $t.setJ(int) :$t.setK(int) : $t.type() @ 2, 3, 2
$t=TriTyp() :  $t.setI(int) : $t.setJ(int) : $t.setK(int) : $t.type() @ 3, 6, 7
```

**Figura 4: População gerada.**

## C. Ferramenta de Teste

A ferramenta JaBUTi [20,21] utiliza o bytecode como base para construir o grafo de fluxo de controle (GFC). A ferramenta distingue as instruções que são cobertas sob execução normal do programa de outros que exigem uma exceção para serem executadas, e por causa disto, o critério todos-nós foi subdividido em dois critérios de teste não-sobrepostos, que são:

- Critério todos-nós independentes de exceção (all-nodes$_{ei}$): requer a cobertura de todos os nós do GFC não relacionados ao tratamento de exceção.
- Critério todos-nós dependentes de exceção (all-nodes$_{ed}$): requer a cobertura de todos os nós do GFC relacionados ao tratamento de exceção.

Analogamente, outros critérios são também subdivididos: critérios todos-ramos independentes de exceção (all-edges$_{ei}$) e todos-ramos dependentes de execeção (all-edges$_{ed}$); todos-usos independentes de exceção (all-uses$_{ei}$) , todos-usos dependentes de exceção (all-uses$_{ed}$), e assim por diante.

## D. Avaliação

O módulo Avaliação verifica a cobertura de cada indivíduo (número de elementos cobertos pelo critério especificado no arquivo de configuração) usando o módulo de avaliação da JaBUTi. Como mencionado anteriormente, cada indivíduo é convertido em uma entrada (um arquivo JUnit) para o programa que está sendo testado pelo Módulo Avaliação. A lista dos elementos cobertos também é salva em um arquivo, pois a análise feita pela JaBUTi pode ficar muito demorada. A matriz apresentada na Figura 5 é obtida e utilizada pela função de fitness que será explicada a seguir.

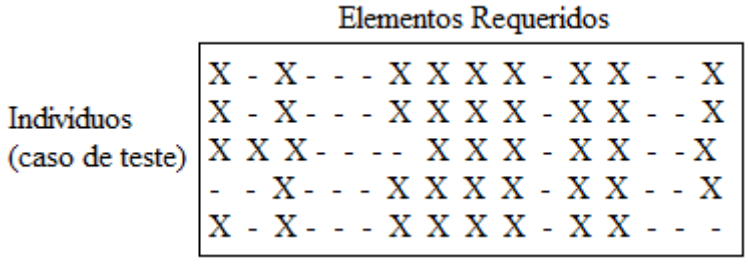


**Figura 5: Informação utilizada para avaliação do fitness.**

## E. Evolução

O Módulo Evolução é responsável pelo processo de evolução e pela aplicação dos operadores genéticos. Os operadores foram implementados com base no trabalho de Tonella [19]. Sendo:

- Mutação: este operador faz alterações, inserções ou remove entradas, construtores, objetos e chamadas de método.
- Crossover: neste operador o ponto de corte pode ser de ambas as partes dos cromossomos, na seqüência de invocações dos métodos ou na parte que contém os parâmetros.

O processo de evolução termina quando o número de gerações é atingido. Alguns mecanismos são utilizados com o objetivo de aumentar o desempenho. Esses mecanismos são ativados pelo testador no arquivo de configuração.

- Fitness: A aptidão de um indivíduo é calculada com base na matriz da Figura 5 e é dada por:

$$\text{Fitness} = \frac{\text{número\_elementos\_cobertos}}{\text{número\_elementos\_requeridos}}$$

Com base na aptidão, os indivíduos são selecionados e os operadores genéticos são aplicados. Nesta versão, TDSGen/OO implementa duas estratégias de seleção: roleta e torneio. As taxas são passadas através do arquivo de configuração.

- Elitismo: Esta estratégia introduz indivíduos na próxima geração com base em uma lista ordenada pelo valor de fitness. Isso garante que os indivíduos com valor de fitness alto estejam na nova população.
- Ineditismo: Esta estratégia tem o objetivo de reduzir o número de indivíduos semelhantes na população (com base na estratégia sharing [4]). Para introduzir os indivíduos na próxima geração, é considerada uma lista ordenada pela métrica de ineditismo. É dado um bônus ao indivíduo que cobre um elemento requerido, não coberto por outros indivíduos da população. Cada elemento i exigido é associado a um bônus de acordo com o número de indivíduos que o cobrem.

$$Bonus_i = 100 \times \left(1 - \frac{nro\_indivíduos\_cobrem}{nro\_total\_individuos}\right)$$

Cada indivíduo x recebe um bônus de ineditismo, que é dado pela soma do bônus para x de cada elemento requerido i.

$$BonusIneditismo_i = \sum_{i=0}^{número\ de\ elementos\ requeridos} Bonus_i$$

## 4. Resultados Experimentais

Para avaliar a ferramenta TDSGen/OO, foi realizado um estudo com quatro programas em Java: TriTyp, Bub, Bisect e Mid. Esses programas foram utilizados por outros autores [14]. Eles são simples, mas podem fornecer uma idéia inicial com relação ao uso da ferramenta.

Foram avaliadas três estratégias, utilizando TDSGen/OO.

a) Geração aleatória (estratégia RA): os dados de teste são obtidos usando a população inicial gerada pelo módulo Geração da População, atribuindo o valor zero ao parâmetro: número de gerações;

b) Geração baseada em Algoritmo Genético (estratégia GA): os dados de teste são obtidos pela desativação dos parâmetros elitismo e ineditismo.

c) Geração baseada na estratégia Ineditismo (estratégia GAU): os dados de teste são obtidos ativando os parâmetros elitismo e ineditismo.

Os critérios todos-ramos$_{ei}$ (AE) e todos-usos$_{ei}$ (AU), implementados pela JaBUTi, foram escolhidos neste estudo, representando, respectivamente, a categoria de critérios baseado em fluxo controle e fluxo de dados.

Os parâmetros elitismo e ineditismo da seção evolução são diferentes de acordo com a estratégia, como explicado anteriormente. Os outros parâmetros foram experimentalmente fixados, exceto o tamanho da população (Figura 6).

{estratégias de evolução}
Taxa de Crossover1: 0.75
Taxa de Crossover 2: 0.75
Taxa de Mutação: 0.75
Tamanho do indivíduo: 20
Tamanho da população: 100
Número de Gerações: 50
Estratégia de Seleção: roleta

**Figura 6: Configuração utilizada no experimento**

Para todos os critérios e estratégias, foram realizadas 5 execuções um valor médio para a cobertura foi obtido. Os resultados são apresentados na Tabela 1.

**Tabela 1: Cobertura obtida (em porcentagem).**

| | *RA* | | *GA* | | *GAU* | |
|---|---|---|---|---|---|---|
| | ***AE*** | ***AU*** | ***AE*** | ***AU*** | ***AE*** | ***AU*** |
| 1-TriTyp | 44 | 39 | 45 | 40 | 56 | 43 |
| 2-Bub | 35 | 26 | 31 | 18 | 75 | 60 |
| 3-Mid | 65 | 73 | 58 | 57 | 88 | 96 |
| 4-Bisect | 30 | 33 | 31 | 40 | 53 | 60 |

## A. Análise dos Resultados

Para uma melhor avaliação, os resultados de cobertura são apresentados em gráficos nas Figuras 7 e 8. Se a cobertura de todos os programas for considerada, as estratégias RA e GA apresentam uma cobertura média semelhante, independentemente do critério. RA apresenta melhor desempenho para os programas Bub e Mid, e GA para os programas TriTpy e Bisect. Nestes casos, a estratégia GA apresenta uma menor cobertura, pois alguns bons indivíduos iniciais são perdidos durante o processo de evolução.

Esse problema não acontece com a estratégia GAU. Aqueles bons indivíduos recebem o bônus ineditismo, e não são descartados facilmente.

Uma explicação para o comportamento da estratégia GA é o menor número de gerações utilizadas (50). Talvez o algoritmo genético pudesse voltar às boas soluções com um número maior de gerações. Mas isso demonstra que o uso do ineditismo realmente contribui para melhorar o desempenho e diminui os custos de execução do Algoritmo Genético.

Outro ponto é que a estratégia GAU sempre apresenta a maior cobertura média, com uma média de 20% de melhora para ambos os critérios.

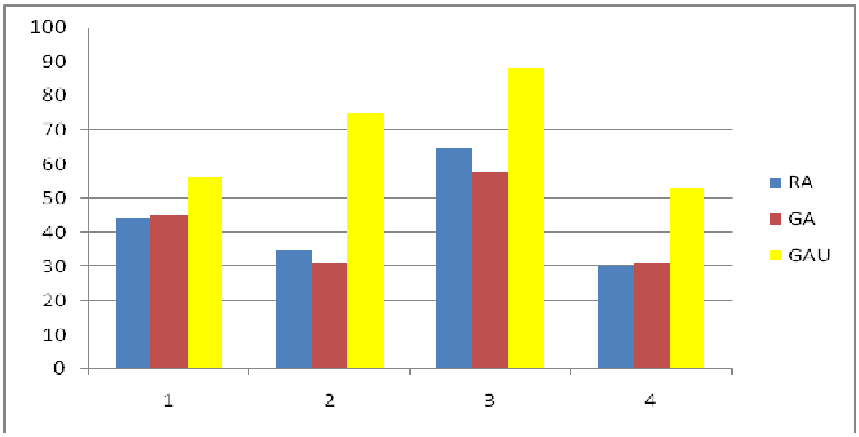


**Figura 7: Comparação de estratégias – Cobertura critério AE.**

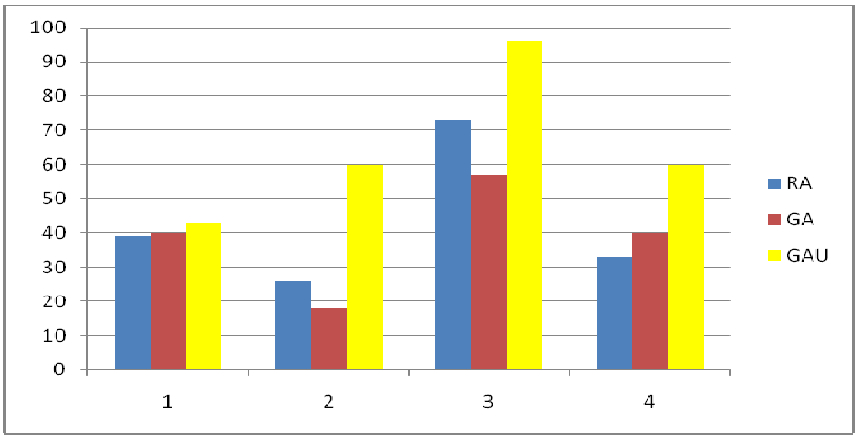


**Figura 8: Comparação de estratégias – Cobertura critério AU**

Observa-se que a cobertura média de AE é muito semelhante à cobertura do critério AU. Ela é maior para os programas Bisect e Bub. Isso é inesperado, pois para satisfazer um critério baseado em fluxo de dados geralmente é mais difícil do que um critério baseado em fluxo de controle. Observa-se também que não há diferença significativa entre os critérios. Este fato pode ser devido ao tamanho dos programas, que são pequenos.

Com relação ao tempo de execução dos algoritmos, a estratégia de RA é a menos custosa. Não há diferença significativa entre as outras estratégias GA e GAU. As médias dos tempos de execução destas estratégias são muito semelhantes, e cerca de 2 vezes o tempo de execução RA.

## 5. Conclusões

Neste trabalho, o framework TDSGen/OO é descrito. TDSGen/OO implementa um Algoritmo Genético para gerar dados de teste no contexto de software orientado a objeto.

O framework tem algumas características importantes que o tornam diferente de outros trabalhos encontrados na literatura. TDSGen/OO trabalha de forma integrada com a ferramenta de teste de JaBUTi, uma ferramenta que permite a aplicação de critérios baseados em fluxo de controle e de dados no teste de unidade de classes Java. Uma característica importante destes critérios é obter os requisitos de teste com base no bytecode e mecanismos de tratamento de exceção.

Desta forma, TDSGen/OO pode ser aplicada mesmo se o código fonte não estiver disponível, pois a função de fitness implementada baseia-se na cobertura dos critérios, fornecida pela ferramenta de teste. Não é necessária qualquer análise adicional ou interpretação do programa em teste.

O fato de estar integrado com a ferramenta JaBUTi permite que o framework seja utilizado em uma estratégia, incluindo diferentes e complementares critérios de teste.

Além disso, com base em um trabalho anterior sobre a geração de dados de teste em código procedural, TDSGen/OO implementa uma estratégia baseada na métrica de

ineditismo, o que parece ser fundamental nos casos em que apenas um caminho particular ou caso de teste cobre um elemento requerido, para manter este teste na população.

No estudo preliminar conduzido, a estratégia GAU (GA + ineditismo) obteve a maior cobertura, sem aumentar o custo. No entanto, outros estudos experimentais devem ser conduzidos. Algumas mudanças (aumento de valor) no número máximo de gerações foram testadas e foi observado um aumento na cobertura para as estratégias GA e GAU. Outro parâmetro que pode influenciar na cobertura obtida de todas as estratégias, incluindo a estratégia aleatória é o tamanho do indivíduo e da população.

Novos experimentos devem avaliar: os parâmetros da seção de avaliação, tais como o número de gerações e aplicação dos operadores genéticos; a eficácia da geração de dados de teste e os custos do algoritmo considerando programas maiores e reais.

Uma limitação observada é a dificuldade de alcançar uma cobertura completa. de A cobertura completa nem sempre é possível devido a elementos não executáveis. Portanto pretende-se incorporar novas funcionalidades ao framework TDSGen/OO para reduzir essas limitações, mas a participação do testador, sempre será necessária.

Um outro trabalho que está sendo realizado é a integração de TDSGen/OO com a ferramenta JaBUTi/AJ, uma versão da JaBUTi que apóia o teste de programas orientados a aspectos, escritos em AspectJ [7], apenas algumas modificações Módulo Ferramentas de Teste são necessárias. Outra possível extensão do framework está relacionada à satisfação de critérios específicos para o teste de integração.

## Agradecimentos

Gostaríamos de agradecer CNPq pelo apoio financeiro.

## Referências

[1] Afzal, W. R. Torkar and Feldt, R. A systematic review of search-based testing for non-functional system properties. Information and Software Technology, vol 51 (6), pp 95 –976. June, 2009.

[2] Cheon, Y.; Kim, M. Y and Peruandla, A. 2005. A Complete Automation of Unit Testing for Java Programs. Disponivel em: http://cs.utep.edu/cheon/techreport/tr05-05.pdf.

[3] Ferreira, L. P. and S Vergilio, S. R.. TDSGEN: An Environment Based on Hybrid Genetic Algorithms for Generation of Test Data. In 17th International Conference on Software Engineering and Knowledge Engineering. Taipei, Taiwan, July., 2005

[4] Goldberg, D. E. and Richardson, J. Genetic algorithms with sharing for multimodal function optimization. In Proc of International Conf. on Genetic Algorithms, 1987.

[5] Harman, M. The Current State and Future of Search Based Software Engineering. International Conference on Software Engineering. pp 342-357. 2007

[6] Harrold, M. J. and Rothermel, G. 1994. Performing data flow testing on claases. In: Second ACM SIGSOFT Symposium on Foundations os Software Engineering, New York: ACM Press, 1994, p. 154-163.

[7] Lemos, O.A.L.; Vincenzi, A.M.; Maldonado, J.C. and Masiero, P.C. Control and data flow structural testing criteria for aspect-oriented programs. Journal of Systems and Softwre, v. 80(6), pp 862-882, June, 2007.

[8] Liu, X.; Wang, B.; and Liu, H. Evolutionary search in the context of object-oriented programs. MIC2005: The Sixth Metaheuristics International Conference, Vienna, Austria. 2005

[9] Mantere, T. and Alander, J.T. Evolutionary software engineering, a review. Applied Software Computing. V. 5., pp 315-331, 2005.

[10] McMinn, P., "Search-based software test data generation: a survey," Software Testing, Verification and Reliability, vol. 14, no. 2, pp. 105–156. 2004.

[11] Michael, C.; McGraw, M. C. and Schatz, M.A. Generating Software Test Data by Evolution. IEEE Transactions on Software Engineering, Vol.SE-27(12): 1085-1110. December. 2001.

[12] Muller, R.A. and Holcombe, M. A Symbolic Java virtual machine for test case generation. In: IASTED Conference on Software Engineering, pp 365-371, 2004.

[13] Offut, A. J. and Irvine, A. Testing object-oriented software using the category-partition method. In: XVII International Conference on Technology of Object-Oriented Languages Systems, p. 293 – 304, Santa Barbara, CA, EUA, Agosto. Prentice-Hall. 1995.

[14] Polo, M.; Piattini, M. and Garcia-Rodriguez, I. Decreasing the cost of mutation testing with second-order mutants. Software Testing, Verification & Reliability, v. 19, pp. 111 – 131, 2009.

[15] Ribeiro, J.C.; Veja, F.F. and Zenha-Rela, M.. Using Dynamic Analisis of Java Bytecode for Evolutionary Object-Oriented Unit Testing. Em 25th Brazilian Symposium on Computer Networks and Distributed Systems, Belém/PA. 2007

[16] Ribeiro, J.C. Search-Based Test Case Generation for Object-Oriented Java Software Using Strongly-Typed Genetic Programming., Em Proc. of the GECCO '08, pp. 1819-1822, Atlanta, Georgia, USA, July 2008.

[17] Sagarna, R., A. and Yao, A. X. Estimation of Distribution Algorithms for Testing Object Oriented Software. In: IEEE Congress on Evolutionary Computation (CEC, 2007).

[18] Seesing and H Gross A Genetic Programming Approach to Automated Test Generation for Object-Oriented Software. In: 1st International Workshop on

Evaluation of Novel Approaches to Software Engineering, Erfurt, Germany, September 19—20, 2006.

[19] Tonella, P. Evolutionary Testing of Classes. ISSTA '04: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software testing and Analysis. ACM Press: 119-128. Boston, Massachusetts, USA, 2004.

[20] Vincenzi, M. R; Delamaro, M. E. e Maldonado, J. C. JaBUTi – Java Bytecode Undestanding and Testing. Technical Report –University of São Paulo, March 2003.

[21] Vincenzi, M. R.; .Delamaro, M.E.; Maldonado, J.C. and Wong, E. Establishing Structural Testing Criteria for Java Bytecode. Software Practice and Experience, 36(14): 1.512 – 1.541. Nov. 2006.

[22] Wapller, S. and Lammermann, F. Using evolutionary algorithms for the unit testing of object-oriented software, Proceedings of the 2005 conference on Genetic and evolutionary computation, June 25-29, Washington DC, USA, 2005.

[23] Wappler, S. and Lammermann, F. Evolutionary Unit Testing of Object-Oriented Software Using a Hybrid Evolutionary Algorithm. In: IEEE Congress on Evolutionary Computation, 2006. CEC 2006. Volume , Issue , pp 851 – 858. 2006

[24] Wappler, S. and Wegener, J.: Evolutionary Unit Testing of Object-Oriented Software Using Strongly-Typed Genetic Programming. Em: GECCO '06', Seattle/ Washignton, 2006.

[25] Wegener, J., Baresel, A. and Sthamer, H. Evolutionary test environment for automatic structural testing. Information and Software Technology, 43:841-854, 2001.