Uma Iniciativa na Definição de uma Estratégia de Teste de Software Combinando Análise Estática e Dinâmica

Cleber Luiz C. Godoy[1], Auri Marcelo R. Vincenzi[1]

[1]Instituto de Informática – Universidade Federal de Goiás (UFGo)
Caixa Postal 131 CEP 74001-970 – Goiânia – GO – Brazil

cleberc.godoy@gmail.com, aurimrv@gmail.com

Abstract. *This study aims to evaluate static analysis tools in open source products of different languages using mutation for this. Initially, only the FindBugs tool, used in Java systems, is being analyzed. With the study is expected to reach a conclusion more concrete of how they may be beneficial in the development of quality software, besides being able to suggest improvements for future versions, and in future studies to make comparisons between different static analysis tools.*

Resumo. *Este trabalho visa avaliar ferramentas de análise estáticas de diferentes linguagens em produtos de código aberto utilizando técnicas de mutação para tanto. Inicialmente, apenas a ferramenta FindBugs, utilizada em sistemas Java, estará sendo analisada. Com o estudo espera-se chegar a uma conclusão mais concreta de como elas podem ser benéficas no desenvolvimento de softwares de qualidade, além de poder-se sugerir melhorias para versões futuras, e em estudos posteriores fazer comparações entre diferentes ferramentas de análise estática.*

1. Introdução

Qualidade de software é algo relevante no contexto de desenvolvimento de software, porém caro e difícil de se alcançar. Diferentes técnicas podem ser usadas para se atingir este fim, inclusive teste, revisão de código e especificação formal [Ayewah 2008]. Dentro desse contexto, Myers define teste de software como sendo o processo de executar um programa com a intenção de encontrar um defeito [Myers 1979]. E para se atingir esse objetivo, existem três técnicas principais de teste: funcional, estrutural e baseado em defeitos.

O teste funcional ou caixa preta avalia o comportamento externo do componente de software. Dados de entrada são fornecidos, o teste é executado e o resultado obtido é comparado a um resultado esperado previamente conhecido. Como detalhes de implementação não são considerados, os casos de teste são todos derivados da especificação. Desse modo, quanto mais entradas são fornecidas, melhor serão os testes. Porém isso também representa um problema, já que é difícil testar todas as entradas possíveis. Outro problema é que a especificação pode estar inconsistente ou ambígua,

fazendo com que as entradas especificadas não sejam as mesmas aceitas para o teste [Pan 1999].

Já o teste estrutural ou caixa branca avalia o comportamento interno do software. Essa técnica trabalha diretamente sobre o código fonte do software para avaliar diversos aspectos tais como: teste de condição, teste de ciclos, dentre outros. Os aspectos avaliados nesta técnica de teste dependerão da complexidade e da tecnologia que determinaram a construção do software. Este tipo de teste é desenvolvido analisando o código fonte e elaborando casos de teste que executem o maior número de possibilidades do produto em teste. Dessa maneira, variações relevantes originadas por estruturas de condições são testadas [Maldonado 2004].

O teste baseado em defeitos consiste em usar o conhecimento sobre os defeitos mais comuns e que se desejam revelar no processo de desenvolvimento do software. Um dos principais critérios deste teste é o Análise de Mutantes. Este critério consiste em utilizar um conjunto de de programas ligeiramente modificados (mutantes) obtidos a partir de determinado programa P para avaliar o quanto um conjunto de casos de teste T é adequado para o teste de P. O objetivo é determinar um conjunto de casos de teste que consiga revelar, por meio da execução de P, as diferenças de comportamento existentes entre P e seus mutantes. Do ponto de vista de teste, cada mutante representa um possível defeito que poderia estar presente no produto em teste [Vincenzi 1998]. Devido a sua natureza, o teste de mutação é usado, frequentemente, como modelo de defeitos, permitindo avaliar a eficácia em detectar defeitos de conjuntos de teste gerados por outros critérios de teste [Andrews *et al.* 2005].

De modo geral, diz-se que o teste de software é uma atividade dinâmica pois demanda a execução ou simulação do produto em teste, sendo possível, desse modo, avaliar o comportamento do produto durante sua execução.

Na área de qualidade de software, destaca-se também a análise estática, que é a técnica na qual se avalia o código fonte sem a necessidade de executá-lo ou considerar uma entrada específica. Para realização desse tipo de análise, pode-se utilizar as ferramentas de análise estática, que ao invés de procurar saber se o código atende às especificações, buscam possíveis violações nas práticas recomendadas de programação, tais como: acesso referências *null*, *overflow* de vetores, divisão por zero, ou ainda sinalizar um problema, como uma comparação que não pode ser verdade, sugerindo que pode ter resultado de um erro de codificação, etc [Nathaniel 2008]. Algumas ferramentas de análise estática existentes são: FindBugs [Hovemeyer e Pugh 2004] e CheckStyle [Jonh 2008] para programas em Java e Splint [Johnson 1977] e CppCheck[Worth 2009] para programas em C e C++, por exemplo.

Desse modo, uma questão que se levanta, é como combinar análise estática e dinâmica de modo a minimizar os custos da atividade de teste.

Para realização desse estudo, foi utilizado teste de mutação em diversos códigos aberto, a fim de simular grande quantidade de defeitos reais [Andrews *et al.* 2005], e posteriormente as ferramentas de análise estática para descobrir o comportamento destas em tais situações. Inicialmente a única ferramenta de análise estática que será analisada é a FindBugs. Uma estrutura para teste então foi criada para realizar os testes e armazenar os resultados.

2. Trabalhos Relacionados

O presente trabalho foi inspirado no trabalho realizado por de Araújo Filho *et al.* [de Araújo Filho *at al.* 2010] que avaliaram a correlação entre defeitos de campo e *warnings* reportados utilizando a ferramenta de análise estática FindBugs. Buscaram respostas para duas questões: se o uso de ferramentas de análise estáticas ajudavam a remover defeitos reportados por usuários e se *warnings* emitidos por ferramentas de análise estática são indícios da existência de defeitos que serão posteriormente reportados. Para a primeira questão, utilizaram 2 sistemas do repositório iBugs (Rhino e ajc) e para a segunda questão utilizaram 25 sistemas da fundação Apache. Obtiveram como resultado que não existe correlação direta entre *warnings* e defeitos de campo, mas que existe uma significativa correlação estatística entre *warnings* e defeitos de campo.

Porém, uma limitação que eles tiveram foi a representatividade da amostra utilizada. O presente trabalho busca superar esse problema usando teste de mutação, uma vez que com esse tipo de teste pode-se gerar uma grande quantidade de defeitos, sendo um modelo muito bom para experimentação, conforme investigado por Andrews *et al.* [Andrews *et al.* 2005].

3. Estrutura para teste

De um modo geral, a estrutura para teste criada envolve: a ferramenta de análise estática FindBugs, que será a única usada inicialmente, um *script* que faz um *parser* da saída desta ferramenta, a ferramenta de mutação MuJava, um banco de dados, um *script* que armazena o resultado da filtragem no banco de dados, um *script* que automatiza a execução da FindBugs e o armazenamento no banco de dados de um programa original e outro *script* que faz o mesmo para os programas mutantes. A Figura 1 mostra um diagrama com a sequência de ações tomadas durante a análise, coleta e armazenamento dos dados.

3.1 Análise da saída da FindBugs

Como o objetivo principal é avaliar a ferramenta como um todo, ela é executada sobre o arquivo JAR de um programa utilizando uma configuração para que sejam reportados *warnings* de todas as prioridades, obtendo assim uma grande quantidade de dados.

Porém faz-se necessária um *parser* de sua saída para facilitar sua armazenagem no banco de dados. Para isso foi criado um *script* usando a linguagem Java que filtra a saída da FindBugs. Uma das saídas disponibilizadas pela FindBugs é no formato XML. Com base nisso, foi usado a API do Java JAXB versão 2.0, e o software Trang para gerar um modelo de objetos em Java correspondente aos possíveis *warnings* gerados pela FindBugs.

A API JAXB do Java é responsável por fazer um *parser* de um documento XML, ou seja, mapear o documento e separá-lo em objetos. Porém para que isso possa ser feito é necessário que se tenha um esquema do documento XML a ser analisado. A ferramenta Trang então, foi responsável por gerar um esquema a partir da saída XML.O comando usando a ferramenta Trang para gerar o esquema da saída foi: “java -jar trang.jar -I xml -O xsd foundbugs.xml findbugs.xsd”, onde “foundbugs.xml” é a saída da FindBugs e “findbugs.xsd” é o nome do esquema gerado.

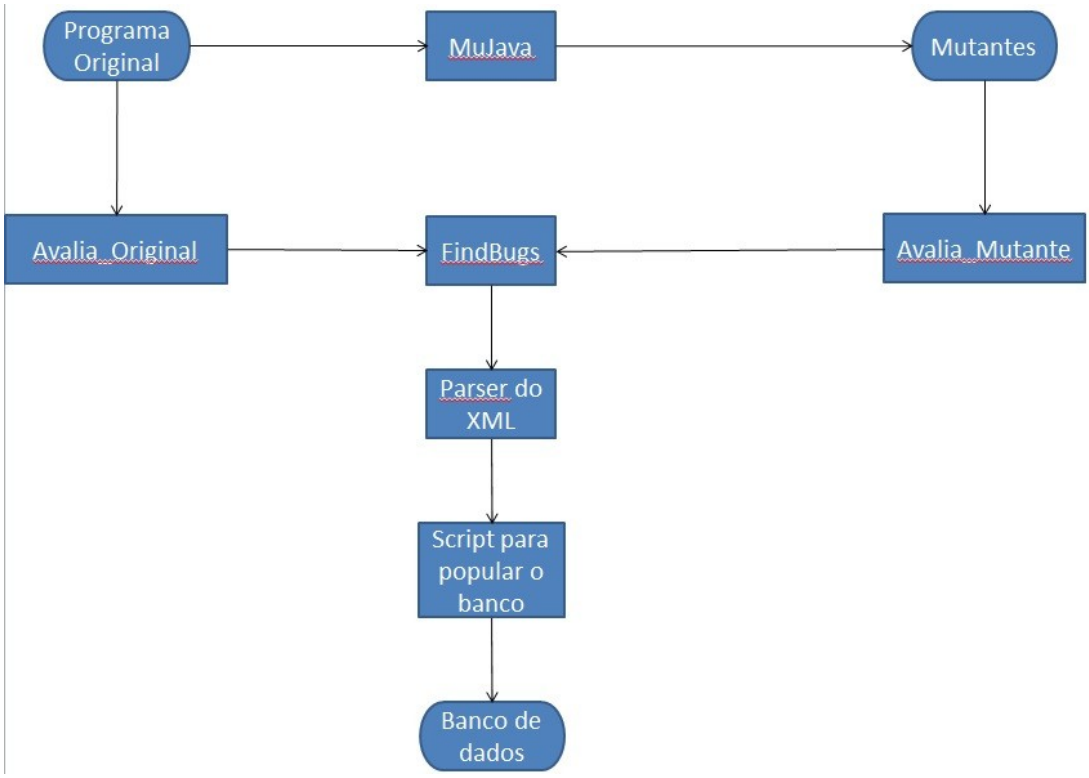

Figura 1. Diagrama com as ações para teste

Feito isso, foi usado a JAXB por criar a partir deste esquema, o modelo de objetos correspondente aos *warnings* da FindBugs. O comando usado para realizar tal ação foi: "xjc -d src -p br.ufg.inf.findbugsParser findbugs.xsd", onde "br.ufg.inf.findbugsParser" é o nome do diretório criado para receber o modelo de objetos criados e "findbugs.xsd", o esquema usado.

3.2 Coleta de dados e armazenamento

Como dito na Estrutura para teste, para realizar a armazenagem de dados, foi desenvolvido um banco de dados. Este foi projetado para que seja genérico, isto é, seja capaz de receber informações de diferentes ferramentas de diferentes linguagens fazendo-se poucas alterações. Foi feito na linguagem MYSQL e utilizou-se a ferramenta MYSQL Workbench para projetá-lo. A Figura 2 mostra o modelo Entidade-Relacionamento do banco de dados criado.

Para fazer a coleta de dados e armazenar os dados no banco, foi desenvolvido um *script* em Java usando a API JDBC. Esta API contém classes e interfaces que permitem o envio de instruções SQL para o banco de dados. Assim, logo após a saída daFindBugs ser mapeada pelo *parser* desenvolvido, os dados dos *warnings* desejados para coleta são selecionados, relacionados com seus respectivos programas e postos no banco de dados.

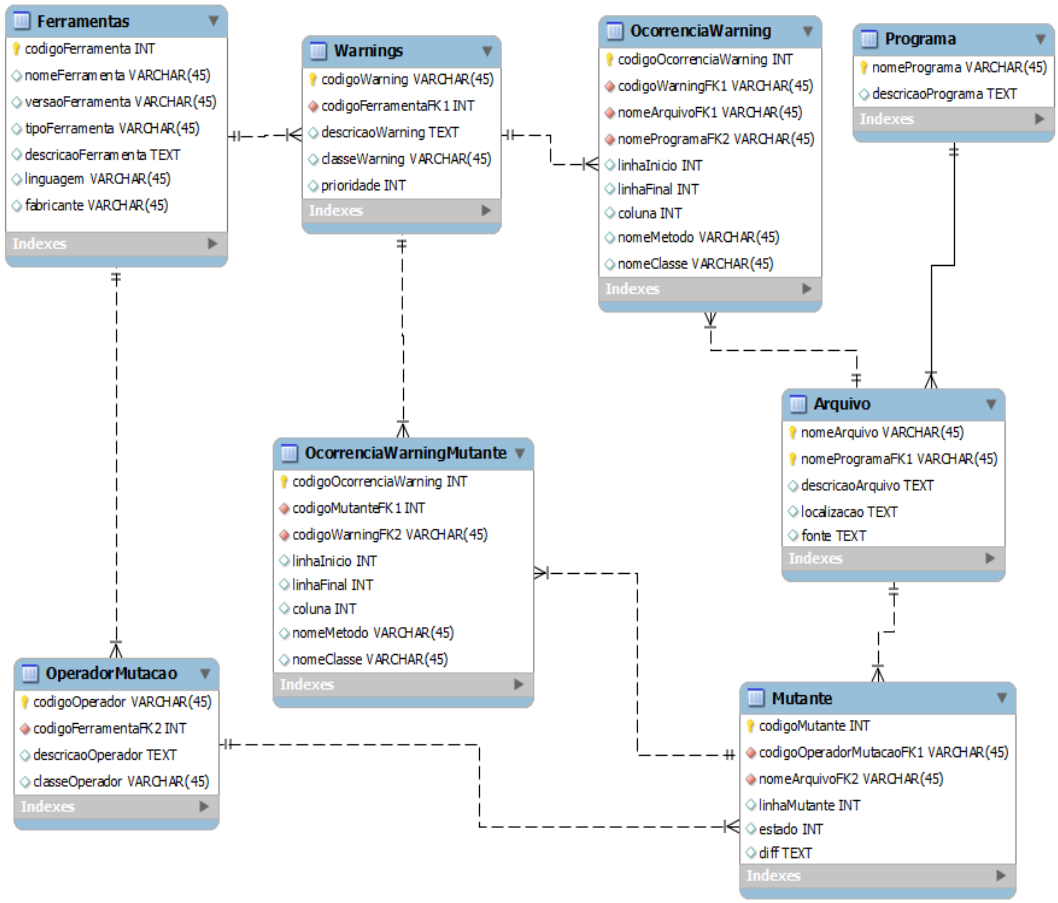

Figura 2: Modelo Entidade-Relacionamento

3.3 Automatização

Apesar de ter-se uma estrutura definida, havia a necessidade de automatizar o processo de análise das saídas da FindBugs e armazenamento no banco de dados devido a grande quantidade de programas e mutantes que deveriam ser analisados. Para isso, criou-se dois *script*s: um para realizar essas tarefas para programas originais e outro para programas mutantes.

Os *script*s foram feitos usando a linguagem Bash. Um deles este executa a FindBugs em um programa original, os *script*s responsáveis pela filtragem e armazenamento dos dados. A Figura 3 mostra o script que avalia um programa original. São passados como parâmetros para o *script* o diretório do programa que se deseja avaliar, o nome do programa e o nome do arquivo padrão segundo a máquina virtual java. Feito isso, o *script* adiciona a extensão ".class", ".java" e ".xml" ao arquivo. Então, é executado o comando para compilar o código fonte do programa original. A partir daí, o comando da FindBugs é executado e depois, o comando para armazenar os *warnings* do programa no banco de dados.

O outro *script* faz o mesmo para um programa mutante, com o diferencial que

este também executa o comando “diff”, que serve para comparar o programa mutante e seu respectivo programa original e detectar o trecho que foi modificado. Assim, pode-se detectar e armazenar essa informação especificadamente.

4. Cronograma e outras atividades

Com os conceitos e *scripts* definidos, o trabalho segue dentro do cronograma. A coleta de dados e armazenamento no banco é a atividade em curso. A partir daí, consultas SQL serão feitas comparando as informações coletadas para que conclusões sobre a eficiência da ferramenta FindBugs possam ser tiradas.

```
#!/bin/bash

if [ $# -lt 1 ]; then
      echo "Uso: $0 <diretorio raiz do programa> <nome
programa> <nome arquivo padrÃ£o JVM>"
else
      DIR=$1
      PROG=$2
      APP=$3
      CLASS=`echo $3 | sed 's/\./\//g'`
      XML=${CLASS}.xml
      JAVA=${CLASS}.java
      CLASS=${CLASS}.class

     CLASSPATH=/home/suporte/NetBeansProjects/findbugsParser/d
ist/findbugsParser.jar:/home/suporte/NetBeansProjects/findbugs
Parser/dist/lib/mysql-connector-java-5.1.15-bin.jar

      cd $DIR

      echo "Compilando o codigo fonte original/$JAVA"
      javac -g -d original original/$JAVA

      echo "Executando a FindBugs em original/$CLASS"
      findbugs -textui -nested:false -effort:max -sortByClass -
low  -jvmArgs "-Duser.language=pt_BR" -xml:withMessages -
output original/$XML original/$CLASS

      echo "Armazendo Warning do programa original localizado
em original/$XML"

      java -cp $CLASSPATH br.ufg.inf.findbugsParser.Main

#     find . -name "*.class" | sed 's/\//./g' | sed 's/..//' |
awk '{print substr($0,0,length($0)-6)}'
fi
```

Figura 3: Script que avalia um programa original

Com a estrutura pronta, posteriormente podem ser feito testes com outras ferramentas, e também de outras linguagens podem ser feitos seguindo a linha de ação determinada no projeto.

As atividades previstas no projeto são listadas abaixo e o cronograma completo é apresentado na Tabela 1. Destaca-se que as atividades estão sendo executadas nos prazos previstos e a vigência da bolsa de iniciação se encerra em outubro de 2011 quando o relatório final será apresentado contendo todos os resultados obtidos.

1. Revisão bibliográfica.
2. Definição do esquema de banco de dados.
3. Definição do filtro para saída da ferramenta de análise estática.
4. Criação de mutantes dos códigos.
5. Armazenamento das saídas da ferramenta no banco de dados.
6. Análise dos resultados.
7. Escrita de artigos e relatórios.

Tabela 1. Cronograma

Ativid.\ Mês	Out. 2010	Nov. 2010	Dez. 2010	Jan. 2011	Fev. 2011	Mar. 2011	Abr. 2011	Mai. 2011	Jun. 2011	Jul. 2011	Ago. 2011	Set. 2011
1												
2												
3												
4												
5												
6												
7												

Referências

Andrews, J. H.; Briand, L. C. & Labiche, Y. Is mutation an appropriate tool for testing experiments? XXVII International Conference on Software Engineering -- ICSE'05, ACM Press, 2005, 402-411.

Myers, G.J, The art of Software Testing. Jonh Wiley & Sons, New York, NY, 1979.

Maldonado, José Carlos; Barbosa, Ellen; Vincenzi, Auri; Delamaro, Márcio; Souza, Simone; Jinor, Mario. Introdução ao teste de software, 2004.

Ayewah, Nathaniel; Hovemeyer, David; Morgenthaler, J. David; Penix, John; e Pugh, William. Using static analysis to find bugs. IEEE Software, 25(5), 2008.

Pan, Jiantao. Software Testing (1999).

Vincenzi, Auri; Subsídios para o Estabelecimento de Estratégias de Teste Baseadas na Técnica de Mutação, 1998.

Hovemeyer, David; e Pugh, William. Finding bugs is easy. SIGPLAN Notices, 2004.

Johnson, S. C.. Lint: A C program checker. Technical Report 65, Bell Laboratories, 1977.

S. F. Jonh. Using Checkstyle with Maven - Pg. 664, 2008.

Worth, D.J., C. Greenough and L.S. Chin. A Survey of C and C++ Software Tools for Computational Science, 2009.

de Araújo Filho, J. E.; de Moura Couto, C. F.; de Souza, S. J. & Valente, M. T. Um Estudo Sobre a Correlação Entre Defeitos de Campo e Warnings Reportados por Uma Ferramenta de Análise Estática. IX Simpósio Brasileiro de Qualidade de Software - SBQS'2010, 2010, 9-23.