Um algoritmo para o cálculo de cobertura de estados

Martim Azevedo do Nascimento e Patrícia Vilain

[1] martim00@gmail.com [2] vilain@inf.ufsc.br

Departamento de Informática e Estatística (INE)
Universidade Federal de Santa Catarina

Abstract. Cobertura de estados é um critério de adequação de testes de software que mede a quantidade de estados modificados durante a execução de um teste que foram cobertos através de asserções. O presente trabalho propõe um algoritmo para o cálculo de cobertura de estados baseado na instrumentação de construções comuns a linguagens orientadas a objetos, como atribuições, retorno de métodos e chamadas de funções. O algoritmo identifica a influência de um atributo no resultado de uma asserção através de um novo cálculo de influências de variáveis em métodos. O artigo apresenta ainda uma implementação através da instrumentação de *bytecode* Java e experimentos utilizando essa ferramenta em um projeto de código aberto.

1 Introdução

Proposta inicialmente por Koster et al [6], cobertura de estados tem como objetivo encontrar testes sem asserções ou asserções fracas. Ao contrário dos critérios de cobertura tradicionais baseados na execução de linhas de código (cobertura de caminhos, cobertura de ramos, entre outras) [10], a cobertura de estados tem como interesse a quantidade de asserções feitas pelo teste que verificam os estados modificados pelo mesmo. Baseia-se na intuição de que um teste pode exercitar todos os caminhos de execução possíveis no programa mas ainda assim não possuir nenhuma verificação do estado resultante. Sem uma asserção nas saídas do programa não é possível garantir com precisão a não existência de erros no programa.

Para demonstrar tal problema considere o código abaixo. Na execução do teste todos os caminhos do programa são executados, atingindo uma adequação de 100% em cobertura de linhas de código. No entanto, a inserção de um erro no programa não irá ser capturada pelo teste pois não há nenhuma verificação no resultado do programa. É esse tipo de problema que a cobertura de estados tenta evidenciar.

```
class Color {
   String name;
   int rgbValue;
   public Color(String name, int rgbValue) {
```

```
      this.name = name;
      this.rgbValue = rgbValue;
   }
   String getName() {
      return name;
   }
}

@Test
public void testWithoutAssert() {
   Color color = new Color("green", 0x00ff00);
   System.out.println(color.getName());
}
```

De maneira geral define-se a cobertura de estados para um teste como sendo a razão entre a quantidade de estados cobertos pela quantidade total de estados modificados.

$$Cobertura\ de\ estados = \frac{Quantidade\ de\ estados\ cobertos}{Quantidade\ total\ de\ estados\ modificados}$$

Considera-se aqui como estado de um programa orientado a objetos o valor dos atributos de suas classes. Um atributo é considerado coberto se influencia alguma asserção dentro do teste, ou seja, se o seu valor é o próprio valor sendo verificado na asserção ou se o mesmo participa da computação do valor sendo verificado. No código abaixo o teste *'testWithAssert'* exercita a classe *'Color'* mostrada anteriormente. No entanto, agora há uma asserção no retorno do método *'getName'*. Como o valor do atributo *'name'* participa da computação do resultado de *'getName'* então o mesmo é considerado coberto.

```
@Test
void testWithAssert()
{
   Color color = new Color("green", 0x00ff00);
   assertTrue(color.getName() == "green");
}
```

É interessante notar que para ser considerado coberto o atributo deve influenciar o resultado da asserção. Não é suficiente que o mesmo tenha seu valor lido na execução de um método sem que tenha influência no seu resultado. Se o método *'getName'* fizesse uma leitura do valor do atributo *'rgbValue'*, como no exemplo abaixo, sem causar influência no retorno o mesmo não poderia ser considerado coberto.

```
String getName() {
   // o atributo rgb no influencia o resultado do mtodo
   System.out.println("Rgb value : " + this.rbgValue);
   return name;
}
```

Cobertura de estados é uma métrica promissora que poderá auxiliar na identificação de testes fracos. No entanto não se conhece nenhum algoritmo com implementação prática, trivial e que utilize construções comuns a linguagens de programação disponível para uso e experimentações.

O presente trabalho propõe um algoritmo com essas características, apresentando um novo cálculo de influências entre variáveis e métodos e demonstrando o seu uso em um projeto de código aberto.

O artigo está organizado da seguinte forma. Na seção 2 são feitas explanações sobre o cálculo de cobertura de estados, apresentando os problemas existentes a serem resolvidos. A seção 3 apresenta a definição do algoritmo de influências de atributos. A seção 4 apresenta a definição do algoritmo de cobertura de estados propriamente dito. A seção 5 mostra algumas especificidades da implementação feita. Na seção 6 são apresentados os resultados do experimento de uso do algoritmo em um projeto de código aberto. Na seção 7 é apresentada uma revisão bibliográfica sobre os trabalhos existentes na área de cobertura de estados. Por fim, na seção 8 são apresentadas as conclusões para o artigo e possibilidades de trabalhos futuros.

2 Cobertura de estados

2.1 Estado de um programa

Cobertura de estados, em linhas gerais, consiste na análise de quais estados do programa foram modificados por um teste e quais desses estados estão sendo verificados por asserções. Mas o que é o estado de um programa? Não há um consenso na literatura de cobertura de estados sobre essa questão. Koster et al [6] definem como estado todas as variáveis vivas no momento da asserção. Isso inclui não só atributos de classes mas também retorno de métodos e parâmetros de métodos passados por referência.

Já Vanoverberghe et al [8] simplificam dizendo que o estado de um programa são os valores dos atributos de suas classes em determinado ponto de execução. Nesta definição, ao exercitar uma classe um teste estará modificando atributos da mesma. Tais atributos serão considerados o estado modificado da classe e portanto deverão ser verificados pelo teste através de asserções.

Neste trabalho optou-se pela definição de Vanoverberghe et al por entendê-la como suficiente para a maioria dos programas orientados a objetos. Mesmo assim o algoritmo proposto já considera o retorno de métodos como construção de interesse para o cálculo de influência e portanto poderá ser facilmente estendido para abranger a definição de Koster et al.

2.2 Modificação de estados

Para calcular a cobertura de estados é necessário saber quando um atributo de um objeto é modificado pelo teste. Para o algoritmo proposto considera-se que os atributos podem ser alterados de duas maneiras:

1. através de atribuições de valores

Exemplo:

```
this.name = "green";
```

2. através da adição e modificação de elementos em estrutura de dados como listas, vetores e dicionários .

Exemplo:

```
private List<String> colors = new ArrayList<String>();
this.colors.add("green");
```

Este último caso é uma sugestão de legibilidade pois apesar da adição de um elemento em uma lista modificar internamente atributos da mesma não fica claro na análise dos resultados de cobertura qual atributo foi realmente modificado. O analisador não tem interesse em saber que um atributo interno à classe java.util.ArrayList, por exemplo, foi modificado pois esse não é o código que está sendo testado. De fato esse é um problema que irá aparecer no uso de qualquer biblioteca de terceiros e o tratamento efetivo para esses casos ficará para trabalhos futuros.

3 Influência de um atributo na asserção

Após conhecer os atributos que foram modificados durante a execução do teste é necessário identificar quais desses atributos foram efetivamente verificados por uma asserção. Para que o atributo seja considerado verificado (e portanto coberto) o seu valor deve ter influência em pelo menos um predicado da asserção. Como dito anteriormente não basta que o seu valor seja lido na execução do predicado mas sim que participe efetivamente do seu resultado. Para resolver esse problema é proposto um algoritmo de verificação de influência de variáveis no retorno de métodos, o qual consiste em adicionar marcações de dependência entre variáveis e dependência entre variáveis e retorno de métodos. Considere o exemplo abaixo:

```
int other() {
  return 10;
}

int m() {
   int x = 0;
```

```
   List<int> l = new ArrayList<int>();
   l.add(x);

   int a = 0;
   int b = a;

   int c = a + other() + l.size();
   return c;
}
```

Como saber que variáveis influenciaram no retorno do método *'m'*? O algoritmo consiste em adicionar marcações em pontos específicos do programa (instruções de atribuição, modificação e verificação de containers e retorno de métodos, por exemplo) declarando relação de dependência entre as variáveis. No exemplo acima as marcações resultantes seriam as seguintes:

```
l <- x       // 'l' depende de 'x' pois o mesmo
             // influencia no seu conteúdo
b <- a       // 'b' depende de 'a'
c <- a       // 'c' depende de 'a'
c <- other() // 'c' depende de 'other'
c <- l       // 'c' depende de 'l' pois uma propriedade
             // de 'l' influencia o seu valor
m() <- c     // 'm' depende de 'c'
```

Ao resolver a relação de dependência para o método *'m'* teríamos que *'m'* depende de *'c'*, que por sua vez depende do método *'other'*, da variável *'a'* e também da lista *'l'*. Como resultado teríamos que *'m'* depende de *'c'*, *'other'*, *'a'* e *'l'* mas não depende de *'b'*. Assim, se uma asserção estivesse verificando o resultado do método *'m'* teríamos que o teste estaria cobrindo além do método *'m'* também as variáveis *'a'*, *'c'*, *'l'* e o método *'other'*. Para este teste a variável *'b'* não estaria coberta. Note que para a definição de estados utilizada no artigo tem-se interesse apenas nos atributos das classes e, portanto, as variáveis locais e o retorno de métodos não entrariam como estados cobertos. No entanto, é necessário fazer a análise dessas dependências pois uma variável local pode ser utilizada como variável temporária na passagem de valores entre dois atributos, como mostra o código a seguir:

```
int m() {
  int a = this.attr1;
  this.attr2 = a;
}
```

Além disso, a análise de variáveis locais serve como demonstração da possibilidade do algoritmo ser estendido para a definição de estados proposta por Koster et al, a qual abrange além de atributos também métodos com retornos e passagem de parâmetro por referência.

4 Definição do algoritmo

O algoritmo proposto está definido em dois passos. O primeiro passo é a instrumentação do código com chamadas de interesse para o cálculo de cobertura. O segundo passo consiste na execução dos testes já instrumentados capturando os dados da execução. Após a execução dos testes, o cálculo de influências é resolvido e o resultado da cobertura de estados é apresentado ao usuário.

4.1 Instrumentação

A instrumentação consiste em inserir três tipos de chamadas entre as instruções do código original: a primeira para realizar o cálculo de influências entre variáveis e métodos, a segunda para observar modificações em atributos e a terceira para evidenciar asserções feitas. Além disso também são inseridas chamadas no início e no final dos métodos de testes para realizar inicializações e finalizações no escopo de execução do teste.

O algoritmo consiste em percorrer as instruções do código sendo instrumentado, buscando construções de interesse para cada tipo de chamada. São elas:

- Atribuições
- Retorno de métodos
- Asserções
- Manipulação de containers
- Verificação de containers

Instrumentação de atribuições.

Definição 1. *Seja A uma atribuição no formato x = y, seja x uma variável, um parâmetro ou um atributo de uma classe e y uma expressão. Seja C o conjunto de identificadores contidos na expressão y. Para cada elemento Ci de C adiciona-se uma dependência de Ci para x.*

Exemplo:

```
public void method() {
   // não tem identificador no lado direito da atribuição
   // então não instrumenta nada
   int a = 0;
   // como 'b' depende de 'a', adiciona-se uma chamada
```

```
   // para registrar essa dependência
   StateCoverage.AddDependency("b", "a");
   int b = a;

   // como 'c' depende de 'b' e 'a' adiciona-se
   // um registro de dependência para cada um
   StateCoverage.AddDependency("c", "a");
   StateCoverage.AddDependency("c", "b");
   int c = b + a;
}
```

Instrumentação de retorno de métodos.

Definição 2. *Seja R o retorno de um método M. Seja E a expressão do retorno R. Seja C o conjunto de identificadores contidos em E. Para cada elemento Ci de C adiciona-se uma dependência de Ci para o método M.*

Exemplo:

```
public int method() {
   int a = 1;
   int b = 2;
   // para cada identificador participante da expressão
   // de retorno adiciona-se uma expressão de dependência
   StateCoverage.AddDependency("method", "a");
   StateCoverage.AddDependency("method", "b");
   return a + b;
}
```

Instrumentação de asserções.

Definição 3. *Seja A uma asserção no formato A(P) em um teste T, aonde A contém um conjunto P de predicados. Para cada elemento Pi de P adiciona-se uma verificação de Pi para o teste T.*

Exemplo:

```
@Test
public void test() {
   Color color = new Color("green", 0x00ff00);

   // para cada asserção encontrado no teste adiciona-se
   // uma chamada para StateCoverage.AddAssert
   StateCoverage.AddAssert("Color.getName()");
```

```
   assertTrue(color.getName() == "green");

   StateCoverage.AddAssert("Color.getRgb()");
   assertEquals(0x00ff00, color.getRgb());
}
```

Instrumentação de containers.

Assume-se que uma classe que possua um atributo container (listas, mapas, conjuntos, etc) pode modificá-lo através da manipulação de seus elementos (inserindo e removendo elementos) e pode verificá-lo através de propriedades do mesmo (como a verificação do seu tamanho ou se contém um determinado elemento).

Modificação de containers .

Para a modificação de containers assume-se as chamadas para métodos modificadores como *List<T>.add* e *List<T>.remove* da biblioteca padrão Java. Como não é possível identificar automaticamente que um método de uma biblioteca é um modificador criou-se um mecanismo de *'whitelist'* para identificar quando uma chamada de método modifica determinado container. Esse *'whitelist'* pode ser estendido pelo usuário adicionando o nome completo do método no formato *<Classe>.<NomeDoMetodo>*. No futuro pretende-se criar um forma mais automatizada para a identificação de métodos de bibliotecas que modificam containers.

Definição 4. *Seja C uma chamada para um método de um container L. Seja W a 'whitelist' contendo os métodos modificadores de L. Se C está contido em W então adiciona-se L como atributo modificado*.

Exemplo:

```
public class Foo {
   private ArrayList<String> list
      = new ArrayList<String>();

   public void method() {
      // adiciona-se o atributo list como modificado
      StateCoverage.AddModification("Foo.list");
      list.add("element");
   }
}
```

Verificação de containers.

Um container é considerado coberto se uma asserção possuir como influência pelo menos uma verificação de propriedade do mesmo. Como verificação de propriedades entende-se a chamada de métodos que apresentam o estado atual do container. Por exemplo, ao adicionar um elemento em uma lista é suficiente que exista uma asserção no tamanho da lista depois da inserção. Nesse caso o uso do método *List<T>.size* como influência em uma asserção serviria como verificação da lista. Novamente aqui utiliza-se o conceito de *'whitelist'* para saber o conjunto de métodos que verificam propriedades em containers.

Definição 5. *Seja A uma asserção no teste T. Seja I o conjunto de influências de A. Seja L uma lista modificada em T. Seja W a 'whitelist' de métodos verificadores de propriedades de containers. Se existir um elemento em I que esteja contido em W então a lista L é considerada coberta em T.*

Exemplo:

```
public class Foo {
   ArrayList<String> list
      = new ArrayList<String>();

   public void getListCount() {
      // o resultado do método "Foo.getListCount()"
      // depende do atributo "Foo.list"
      StateCoverage.AddDependency("Foo.getListCount()"
         , "Foo.list");
      return list.size();
   }
}

@Test
public void testFooIsEmpty() {
   Foo foo = new Foo();
   // ao verificar o resultado de "Foo.getListCount()"
   // o teste estará cobrindo automaticamente
   // o atributo "Foo.list", pois "Foo.getListCount()"
   // depende de "Foo.list()"
   StateCoverage.AddAssert("Foo.getListCount()");
   assertEquals(0, foo.getListCount());
}
```

4.2 Execução dos testes e resolução do cálculo de influências

Após a execução dos testes, as informações coletadas são processadas por um solucionador que a partir das asserções feitas resolve recursivamente a cadeia de dependências da mesma, resultando em um conjunto de influências. Os atributos que estiverem contidos neste conjunto são considerados cobertos.

Definição 6. *Seja A(P) uma asserção feita no teste T. Seja P um predicado de A. Seja D o conjunto de influências de P. Para cada influência Di de D adiciona-se Di no conjunto resultante R. Percorre-se recursivamente as influências de Di adicionando-as também no conjunto R. R será o conjunto de influências da asserção.*

Definição 7. *Seja A uma asserção feita no teste T. Seja attr um atributo modificado em T. Seja R o conjunto de influências de A. Se attr estiver contido em R então o mesmo é considerado um estado coberto em T.*

4.3 Cálculo da cobertura de estados

A cobertura de estados é calculada para cada teste isolado e para a suíte como um todo. Para cada teste é feito a contagem de estados modificados pelo mesmo e os estados verificados pelas asserções. A taxa de cobertura de estados se dará pela divisão entre a quantidade de estados verificados pela quantidade de estados modificados.

Já para a suíte como um todo, utiliza-se a mesma definição otimista proposta por Koster et al [6], aonde a cobertura de estados para a suíte se dá pela união entre todos os estados verificados por asserções dividido pela união entre os estados modificados pela suíte.

5 Implementação

Foi realizada uma implementação do algoritmo usando instrumentação do *bytecode* java, através da biblioteca ASM [4]. Para isso foi utilizada a estrutura de visitadores de *bytecode* disponíveis na biblioteca. Nela é possível instrumentar um arquivo *.class* apenas reimplementando métodos da classe *'ClassVisitor'*. Quando um método é encontrado no arquivo *.class* a biblioteca chama o método *'visitMethod'* da classe *'ClassVisitor'* aonde é possível retornar um implementação de *'MethodVisitor*' afim de instrumentar as instruções contidas no *bytecode* do método. A biblioteca também disponibiliza classes para realizar análise de fluxo de dados diretamente nas instruções de um método. Para isso é necessário estender a classe *'Interpreter'*, reimplementando métodos que serão chamados quando instruções de determinados tipos foram encontradas no *bytecode*. Por exemplo, quando uma instrução que movimenta valores entre a pilha e variáveis locais é encontrada o método abstrato *'copyOperation'* da classe *'Interpreter'* é chamado. É o caso de instruções como a *'xLOAD*' e *'xSTORE'*. Quando há uma chamada de um método ou uma interface o método abstrato *'naryOperation'* é chamado. É o caso de instruções como *'INVOKEVIRTUAL'* e *'INVOKESTATIC'*.

Utilizando essa estrutura foi possível realizar a instrumentação do código com algumas chamadas de interesse para o cálculo de cobertura de estados. A implementação está disponível como código aberto [2].

6 Resultados

Foram feitos experimentos com a ferramenta no projeto Tablelize-it, um projeto de código aberto contendo uma suíte de testes unitários. [3]. O experimento consistiu inicialmente em medir a taxa de cobertura de linhas de código e a taxa de cobertura de estados para o projeto em questão. A Figura 1 mostra a relação entre as duas métricas de cobertura.

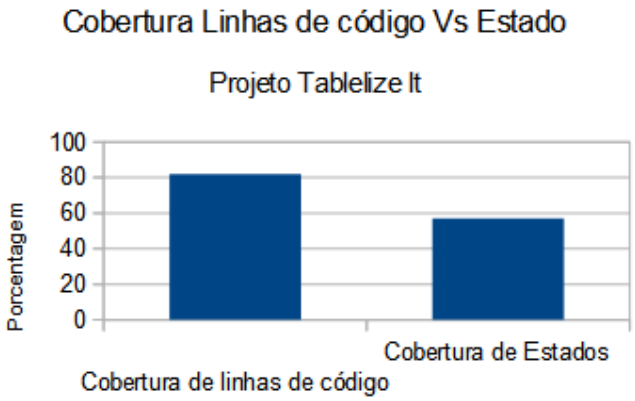

Fig. 1. Taxa de cobertura de linhas de código X cobertura de estados para o projeto Tablelize-it

Através da ferramenta EclEmma [1] identificou-se uma taxa de cobertura de 81,8% de linhas de código para o projeto. No entanto, ao executar a mesma suíte de testes, agora medindo cobertura de estados com a ferramenta proposta no artigo, atingiu-se apenas 57% de cobertura, ou seja, dos atributos modificados pelos testes apenas um pouco mais da metade deles foram verificados por asserções.

Na sequência foi modificado um desses atributos não cobertos pelo teste (segundo a cobertura de estados) e inserido nele um *'bug'* propositalmente. O atributo modificado foi o *'positionInFile'* da classe *'Table'*, como mostra a Figura 2. A linha modificada está em amarelo e consistiu no incremento do atributo em uma unidade.

```
public class Table {

    // ...

    private int positionInFile = 0;

    public Table(String tableName) {
        this.tableName = tableName;
    }

    public void setPositionInFile(int positionInFile) {
        this.positionInFile = positionInFile;
        this.positionInFile++;
    }

    // ...
```

Fig. 2. Inserção de um bug no atributo *'positionInFile'* da classe *'Table'*

Após executar novamente a suíte de testes notou-se que a mesma continuou passando, sem evidenciar o erro inserido.

É interessante notar que tanto a inicialização do atributo quanto suas modificações estavam sendo executadas pelo teste (como ilustra a Figura 3), ou seja, estavam adequadas quanto ao critério de cobertura de linhas de código. Isso demonstra que a simples execução de uma linha de código pelo teste não é suficiente para demonstrar a sua validade.

```
public class Table {

    // ...

    private int positionInFile = 0;

    public Table(String tableName) {
        this.tableName = tableName;
    }

    public void setPositionInFile(int positionInFile) {
        this.positionInFile = positionInFile;
        this.positionInFile++;
    }

    // ...
```

Fig. 3. Relatório de cobertura de linhas de código apresentado pela ferramenta EclEmma para a classe *'Table'*

7 Trabalhos existentes

Koster et al [6] foram os primeiros a definir cobertura de estados e cunhar o termo. Em [6] os mesmos definem cobertura de estados como sendo a razão entre as saídas de um programa que são verificadas por asserções e o total de saídas do programa. Como saída de um programa entende-se os últimos valores definidos das variáveis que se mantém vivas no momento das asserções. Isso exclui, por exemplo, variáveis locais mas mantém atributos de classes e retorno de métodos. Para cada saída o último *'statement'* que a definiu é considerado o seu *'Output-defined statement'* ou ODS.

Para o cálculo da cobertura de estados é proposto a utilização de *'program slice'* [9]. Se o *'slice'* de uma variável que seja uma saída do programa a partir da asserção contém o ODS da variável então a mesma estará coberta.

Em Koster [5] é apresentada uma nova forma de calcular cobertura de estados através do uso de *'dynamic taint analysis'* [7]. Nessa abordagem os ODS são considerados os *'sources'* da análise e as asserções os *'sinks'*. Se uma variável é definida em um *'source'* mas não chega em nenhum *'sink'* então ela é considerada como não coberta e portanto inadequada quanto a cobertura de estados. Caso contrário, ela estará coberta.

Vanoverbergh et al [8] propõem uma outra definição para o cálculo de cobertura de estados restringindo a noção de estados de um programa como sendo os atributos das classes que o compõem. Nessa abordagem é proposto um monitoramento nas modificações e nas leituras de atributos afim de verificar que estados foram modificados e quais foram verificados por asserções durante a execução do teste. Para saber se um atributo foi verificado por uma asserção é utilizado um monitoramento de quando a execução está dentro da asserção. Neste momento todos os atributos lidos e que influenciam no resultado da mesma são considerados cobertos.

Apesar de citarem o uso de algoritmo de fluxo de dados como forma de verificar a influência de atributos no resultado de uma asserção, não há muitos detalhes sobre como acontece o seu uso. Além disso o algoritmo pressupõe uma estrutura específica de monitoramento de atributos e asserções. Não fica claro, no entanto, como tal estrutura se refletiria em uma implementação do algoritmo em uma linguagem de programação.

8 Conclusões

O presente trabalho apresentou um novo algoritmo para cobertura de estados, utilizando-se de instrumentação de construções comuns a linguagens de programação orientadas a objetos. Foi possível concluir que uma implementação do algoritmo é viável na prática, mostrando resultados promissores tanto em eficiência quanto em praticidade de uso.

Experimentos feitos em um projeto de código aberto permitiram encontrar atributos que foram modificados pelos testes mas para os quais não haviam nenhuma asserção. Ao inserir um erro proposital em um desses atributos notou-se que a suíte de testes continuou passando, evidenciando o problema.

É notável a validade de cobertura de linhas de código como forma de guiar o testador na busca de caminhos não testados e de nenhuma maneira ela pode ser ignorada. No entanto, a cobertura de estados se apresenta como um critério de adequação complementar que vem auxiliar ainda mais o testador na árdua tarefa de validar um programa.

Como trabalhos futuros espera-se melhorar a integração com bibliotecas de terceiros com o objetivo de identificar de forma mais automatizada possíveis métodos que modificam objetos sem precisar fazer uso de *'whitelists'*.

Além disso pretende-se incluir outros tipos de construções como estado de um programa, como por exemplo o retorno de métodos públicos e parâmetros passados por referência, como definido em Koster et al [6].

Referências

1. EclEmma. Java Code Coverage for Eclipse. `http://www.eclemma.org/`. Acessado em 2014-02-20.
2. Scova. A java implementation of state coverage. https://github.com/martim00/sc. Acessado em 2014-02-20.
3. Tablelize-it. A simple framework for scaling unit tests by example. https://github.com/martim00/tablelize_it. Acessado em 2014-02-20.
4. Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.
5. Ken Koster. A state coverage tool for JUnit. In *Companion of the 13th international conference on Software engineering - ICSE Companion 2008*, page 965. Association for Computing Machinery, 2008.
6. Kenneth Koster and David C. Kao. State coverage. In Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering - ESEC-FSE 2007, page 541. Association for Computing Machinery, 2007.
7. Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *2010 IEEE Symposium on Security and Privacy*, pages 317–331. Institute of Electrical and Electronics Engineers, May 2010.
8. Dries Vanoverberghe, Jonathan Halleux, Nikolai Tillmann, and Frank Piessens. State Coverage: Software Validation Metrics beyond Code Coverage. pages 542–553. Springer-Verlag, 2012.
9. Mark Weiser. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.
10. Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, Dec 1997.