Teste de Unidade de Programas Orientados a Aspectos

Otávio Augusto Lazzarini Lemos[1] *, Auri Marcelo Rizzo Vincenzi[2],
José Carlos Maldonado[1] e Paulo Cesar Masiero[1]
[1]Universidade de São Paulo, Instituto de Ciências Matemáticas e de Computação,
Av. do Trabalhador São-Carlense, 400, São Carlos, SP
[2]Centro Universitário Eurípides de Marília
Av. Hygino Muzzi Filho, 529, Cx. Postal 2041, 17525-901 Marília, SP
e-mail: [oall,jcmaldon,masiero]@icmc.usp.br, auri@fundanet.br

Resumo

Neste artigo é apresentada uma abordagem de teste estrutural de unidade para programas orientados a aspectos. Um modelo de fluxo de controle orientado a aspectos é definido a partir do bytecode (código objeto de Java) resultante do processo de compilação/combinação de programas escritos na linguagem AspectJ – uma extensão da linguagem Java para a programação orientada a aspectos – e critérios de teste são derivados desse modelo. Além dos dois critérios de teste estruturais tradicionais todos-nós e todas-arestas, dois outros critérios de teste específicos para programas orientados a aspectos, todos-nós-de-interceptação e todas-arestas-de-interceptação, são definidos. Um exemplo de aplicação dos critérios é apresentado.

Abstract

This paper presents a structural unit testing approach for aspect-oriented programs. An aspect-oriented control flow model is defined based on the bytecode (the object code of Java) resulted from the compilation/weaving of programs written in AspectJ – an extension of the Java language for the aspect-oriented programming – and testing criteria are derived from this model. Besides the two traditional structural testing criteria all-nodes and all-edges, two other aspect-oriented specific structural testing criteria, all-interception-nodes and all-interception-edges, are defined. An example of the application of the criteria is presented.

1. Introdução

A Programação Orientada a Aspectos (POA) é uma nova técnica de programação concebida para permitir o uso mais efetivo do princípio da separação de interesses no desenvolvimento de software. Até agora, a maioria das discussões na área tem focado nos conceitos básicos da POA, em propostas de implementações de linguagens orientadas a aspectos e em diversos tipos de aplicação. Recentemente os pesquisadores estão se preocupando também com outros problemas relacionados com o desenvolvimento de software orientado a aspectos (*aspect-oriented software development*), tais como técnicas de projeto e abordagens de verificação e teste de software [1]. A POA tem sido reconhecida como uma técnica que permite a construção de sistemas com melhor arquitetura, facilitando a manutenção dos diferentes interesses e a legibilidade do código. Entretanto, a sua simples utilização não evita que erros sejam introduzidos ao longo do desenvolvimento do software e dessa maneira técnicas de verificação, validação e teste (VV&T) continuam sendo importantes no processo de desenvolvimento de software orientado a aspectos [17].

*Com auxílio financeiro do Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq).

O teste estrutural (também conhecido como teste *caixa-branca*) deriva seus requisitos a partir da estrutura lógica dos programas. A principal motivação da técnica é que não se pode confiar em um trecho de um programa se ainda existem certos caminhos que nunca foram executados. Dois exemplos de critérios de teste estrutural baseados no fluxo de controle da aplicação são: todos-nós e todas-arestas [12].

Até o presente momento, Zhao [17, 18] foi o único pesquisador a apresentar uma abordagem de teste estrutural de programas orientados a aspectos. Em seus artigos, Zhao propõe uma abordagem de teste de unidade baseada no fluxo de dados para testar programas orientados a aspectos com base na linguagem AspectJ – uma das linguagens orientada a aspectos mais utilizadas na atualidade [9]. Na época em que o pesquisador escreveu esses artigos, os modelos de fluxo de controle e de dados que podiam ser derivados dos programas escritos em AspectJ tornavam o teste estrutural mais complexo. O problema é que a solução adotada pelo AspectJ naquela época era de inserir código relacionado com os aspectos diretamente nos pontos onde estes interceptavam os componentes e assim os módulos tratados no teste misturavam código de aspectos com código de componentes.

Neste artigo é apresentada uma abordagem de teste estrutural baseada na mais recente estratégia de implementação adotada pelo AspectJ [8] e na extensão do trabalho de Vincenzi [15] – que propõe modelos e critérios de teste para programas escritos em Java. A principal idéia é que baseado na estratégia de implementação do AspectJ e trabalhando com os artefatos resultantes dos processos de compilação/combinação (*weaving*), pode-se ter uma abordagem que implica em uma aplicação muito mais direta das conhecidas técnicas de teste estruturais propostas para programas procedimentais e orientados a objeto. Dessa maneira tem-se o benefício de que muitos resultados obtidos para esses paradigmas também podem ser levados em conta no contexto da POA e torna-se possível a utilização de ferramentas tradicionais com poucas extensões. Outro ponto importante é que a separação de interesses obtida nas primeiras fases do desenvolvimento de software pode ser mantida na atividade de teste, isto é, os aspectos podem ser testados separadamente das classes afetadas. Além disso, também podem ser definidos critérios de teste estruturais específicos para programas orientados a aspectos.

Na Seção 2 a Programação Orientada a Aspectos é introduzida brevemente e são fornecidos os principais conceitos da linguagem AspectJ, além da sua estratégia de implementação. Na Seção 3 são abordados os principais conceitos da técnica estrutural e como aplicá-la no teste de unidade de programas orientados a aspectos. Na Seção 4 são definidos critérios de teste de fluxo de controle para programas orientados a aspectos. Na Seção 5 é apresentado um exemplo da aplicação da abordagem e, por fim, na Seção 6 são apresentadas as principais conclusões e trabalhos futuros.

2. Programação Orientada a Aspectos e a Linguagem AspectJ

Em um projeto de software – para fins de simplicidade, legibilidade, e conseqüente facilidade na manutenção e maior potencial para reutilização – é importante que os vários interesses referentes ao sistema estejam localizados em módulos separados. De forma geral, todas as técnicas de programação oferecem suporte a esse tipo de *separação de interesses*, porém cada uma a sua maneira (utilizando por exemplo sub-rotinas, procedimentos, funções, classes, APIs) e em graus diferentes.

Em meados dos anos 90, alguns pesquisadores constataram a existência de certos interesses que, independentemente da técnica de programação utilizada ou da maneira como o sistema é decomposto, não se encaixam em módulos individuais, mas ficam espalhados por várias unidades do software. Esses tipos de interesse são geralmente não funcionais, e também são chamados de interesses *ortogonais*, pois podem ser visualizados como integrantes de uma segunda di-

Tabela 1: Tipos de pontos de corte do AspectJ e suas respectivas sintaxes (adaptado de [10]).

Tipo	**Sintaxe**
Execução de método	execution(AssinaturaDeMétodo)
Chamada a método	call(AssinaturaDeMétodo)
Execução de construtor	execution(AssinaturaDeConstrutor)
Chamada a construtor	call(AssinaturaDeConstrutor)
Iniciação de classe	staticinicialization(AssinaturaDeTipo)
Acesso de leitura de atributo	get(AssinaturaDeAtributo)
Acesso de modificação de atributo	set(AssinaturaDeAtributo)
Execução de tratador de exceção	handler(AssinaturaDeTipo)
Iniciação de objeto	initialization(AssinaturaDeConstrutor)
Pré-iniciação de objeto	preinitialization(AssinaturaDeConstrutor)
Execução de advices	adviceexecution()

mensão, ortogonal àquela dos módulos que normalmente implementam requisitos funcionais (os componentes). Exemplos de possíveis interesses ortogonais são: políticas de sincronização, mecanismos de tolerância a falhas, funcionalidades de qualidade de serviços (*QoS*), controle de acesso, rastreamento da execução de programas e persistência [3].

A POA oferece mecanismos para que esses interesses ortogonais sejam implementados em módulos separados e fornece meios para a definição de pontos do programa onde esses módulos (os chamados aspectos) possam adicionar comportamento. Assim, a POA pretende dar suporte aos interesses ortogonais da mesma maneira que a Programação Orientada a Objetos (POO) tem dado suporte aos objetos [9].

Uma linguagem orientada a aspectos de propósito geral deve determinar: 1) um modelo de pontos de junção (pontos bem definidos na execução de um programa [14]), que descreve os ganchos nos componentes onde as interceptações podem ocorrer; 2) um mecanismo de identificação desses pontos de junção; 3) unidades que encapsulam especificações de pontos de junção e mudanças de comportamento desejados (os aspectos); e 4) um processo para combinar os componentes com os aspectos em um programa (combinação) [3].

2.1. A Linguagem AspectJ

A linguagem AspectJ é uma extensão de Java criada para permitir a programação orientada a aspectos de maneira genérica. Basicamente, as novas construções do AspectJ consistem em: pontos de corte (`pointcut`) que identificam conjuntos de pontos de junção; pré-sugestões, pós-sugestões e sugestões substitutivas (`before`, `after` e `around` *advices*) que definem as alterações no comportamento antes, após ou ao invés de um ponto de junção; construções para afetar estaticamente a estrutura dos módulos básicos do programa (declarações inter-tipos e que alteram a hierarquia de classes); e os aspectos em si (`aspect`). Os aspectos combinam especificações de pontos de junção (utilizando os `pointcuts`) e sugestões que são as próprias alterações de comportamento, além de métodos e atributos.

O modelo de ponto de corte do AspectJ é baseado em diferentes eventos que podem ocorrer nos componentes. Na Tabela 1 são mostradas todas as categorias de pontos de junção disponíveis no AspectJ e as sintaxes dos pontos de corte associados. Por exemplo, o desenvolvedor do aspecto pode querer adicionar comportamento toda vez que um método retorna ao ponto de sua chamada. Para implementar esse tipo de funcionalidade no AspectJ, ele poderia utilizar uma pós-sugestão (`after advice`) em um ponto de corte do tipo `call` no método desejado.

Na Figura 1 é listada parte do código fonte de uma aplicação orientada a aspectos para gráficos

2D, que será utilizada ao longo deste artigo. A classe `Point` implementa pontos com coordenadas x e y. Já a classe `LineSegment` implementa segmentos de linha compostos de dois pontos – o início e o fim – e possui, entre outros, um método para calcular o ponto de intersecção entre dois segmentos. O aspecto `FigureLogging` imprime mensagens indentadas quando métodos do pacote *component* são chamados (exceto métodos `get*`). Além disso, informa qual é a classe dona do método – se é `Point`, `LineSegment` ou alguma outra. O atributo `indentationLevel` é utilizado para imprimir um número de espaços correspondente à profundidade da chamada ao método que está sendo chamado. Antes de imprimir a mensagem de registro, o nível de indentação (`indentationLevel`) é incrementado de um, sendo decrementado no retorno da chamada.

2.1.1. Estratégia de Implementação. Em qualquer linguagem orientada a aspectos, o código dos aspectos e dos componentes deve executar de maneira coordenada a fim de refletir a semântica da linguagem. Para isso um ponto importante é assegurar que as sugestões sejam executadas nos pontos de junção apropriados. As versões anteriores do AspectJ utilizavam a estratégia de inserir todo o bloco de código referente às sugestões diretamente nos pontos de junção, o que resultava em arquivos **`.class`** que continham tanto código de aspectos quanto código de componentes. Dessa maneira, não havia como distinguir as partes dos aspectos das partes dos componentes a partir do código objeto. Na verdade, antigas versões do compilador do AspectJ combinavam o código dos aspectos com o código dos componentes no próprio código-fonte. O combinador do AspectJ 1.1.1 é baseado no bytecode e portanto esse processo é feito por transformação do bytecode ao invés de diretamente no código-fonte[1] [8].

O combinador de sugestões transforma estaticamente os programas de maneira que em tempo de execução ele se comporta de acordo com a semântica da linguagem. O compilador aceita tanto bytecode Java quanto código-fonte e produz bytecode Java. A idéia principal é compilar as declarações de aspectos e sugestões em classes e métodos Java comuns (em bytecode). Parâmetros passados às sugestões se transformam em parâmetros desses métodos (com algum tratamento especial quando informações reflexivas são necessárias). Para coordenar os aspectos com os componentes o bytecode é instrumentado e chamadas às sugestões são inseridas considerando que certos locais no bytecode representam possíveis pontos de junção. Além disso, se o ponto de junção não pode ser completamente determinado em tempo de compilação, essas chamadas são englobadas por testes dinâmicos para ter certeza de que as sugestões são executadas somente nos momentos apropriados (esses testes são chamados de *resíduos*) [8].

Com essa estratégia de implementação é possível identificar as interceptações feitas pelos aspectos (chamadas aos "métodos sugestão") no bytecode resultante da compilação. Isto é, toda chamada que é feita a uma sugestão no bytecode é na verdade uma interceptação da sugestão correspondente de algum aspecto. A partir desses resultados um modelo de fluxo de controle baseado no bytecode pode representar completamente o fluxo de controle do programa orientado a aspectos de maneira a aplicar a técnica estrutural em cada módulo.

Na Figura 2 é mostrado uma parte do bytecode resultante do método `distance` da classe `LineSegment` que calcula a menor distância entre a linha que contém o segmento de linha e um ponto. As interceptações são identificadas pelas chamadas aos métodos (sugestões) `ajc$before$aspects_FigureLogging$1$114ea88f` (instruções em 221 e 245) e `ajc$afterReturning$aspects_FigureLogging$2$114ea88f` (instruções em 230 e 254) gerados pelo AspectJ. Tais métodos correspondem às pré e pós sugestões do aspecto `FigureLogging`, como se pode deduzir dos próprios nomes dos métodos. Um

[1] O compilador de AspectJ pode utilizar a antiga estratégia em casos especiais mas a opção `-XnoInline` o previne de fazê-lo [8].

```
package components;
public class Point implements
  FigureElement {
 private double x = 0, y = 0;
 public Point(double _x, double _y) {
   x = _x;
   y = _y;
 }
 public void setX(double _x) {
   x = _x;
 }
 public void setY(double _y) {
   y = _y;
 }
  ...
 public void rotate(Point center, double
   angle ) {
   double radius = this.distance(center);
   double deltaY = y - center.getY();
   double deltaX = x - center.getX();
   double theta =
     Math.atan(deltaY/deltaX );
   x = center.getX() +
     radius * Math.cos(angle+theta);
   y = center.getY() +
     radius * Math.sin(angle+theta);
 }
}

package components;
public class LineSegment implements
  FigureElement {
  private Point p1, p2;
  ...
  public Point intersection(LineSegment
   l2) {
   double x1  = this.getP1().getX();
   double y1  = this.getP1().getY();
   double x2  = this.getP2().getX();
   double y2  = this.getP2().getY();
   double x3  = l2.getP1().getX();
   double y3  = l2.getP1().getY();
   double x4  = l2.getP2().getX();
   double y4  = l2.getP2().getY();
   double uaN = (x4 - x3)*(y1 - y3) -
     (y4 - y3)*(x1 - x3);
   double ubN = (x2 - x1)*(y1 - y3) -
     (y2 - y1)*(x1 - x3);
   double den = (y4 - y3)*(x2 - x1) -
     (x4 - x3)*(y2 - y1);
   if (den == 0)
     return null;
     // coincident or parallel lines
   else {
     double ua  = uaN/den;
     double ub  = ubN/den;
     double x = x1 + ua * (x2 - x1);
     double y = y1 + ua * (y2 - y1);
     FigureFactory ff = new
       FigureFactory();
     Point p = ff.makePoint(x, y);
     return p; // intersection
   }
 }
 double distance(Point pt) {
   double t, bot, dot, xn, yn;
   if(pt == null)
     return java.lang.Double.POSITIVE_INFINITY;
   bot = java.lang.Math.pow((p2.getX() -
     p1.getX()),2) +
     java.lang.Math.pow((p2.getY() -
     p1.getY()),2);
   if ( bot == 0.0 ) {
     xn = p1.getX();
     yn = p1.getY();
   } else {
     dot = (pt.getX() - p1.getX()) *
     (p2.getX() - p1.getX()) +
     (pt.getY() - p1.getY()) *
     (p2.getY() - p1.getY());
     t = dot / bot;
     xn = p1.getX() + t * (p2.getX() -
          p1.getX());
     yn = p1.getY() + t * (p2.getY() -
          p1.getY());
   }
   FigureFactory ff = new FigureFactory();
   Point Pn = ff.makePoint(xn,yn);
   double dist = pt.distance(Pn);
   return dist;
 }
}

package aspects;
public aspect FigureLogging {
 declare precedence: FigureLogging,
   PointBoundsChecking, DisplayUpdating, *;
 private int indentationLevel = 0;
 public pointcut loggedOperations()
 : call(* components.*.*(..)) &&
   !call(* components.*.get*(..));
 before() : loggedOperations() {
  indentationLevel++;
  Signature sig =
    thisJoinPointStaticPart.getSignature();
  if (thisJoinPointStaticPart.getSignature().
      getDeclaringType().getName()
      == "components.LineSegment")
     PrintIndented("LineSegment <" +
       sig.getName() + ">");
  else
    if (thisJoinPointStaticPart.getSignature().
        getDeclaringType().getName()
        == "components.Point")
     PrintIndented("Point <" + sig.getName()
       + ">");
    else
       PrintIndented("other <" +
         sig.getName() + ">");
 }
 after() returning(): loggedOperations() {
   indentationLevel--;
 }
 void PrintIndented(String s) {
   for (int i = 0,
     spaces = indentationLevel * 4;
     i < spaces; ++i) {
      System.out.print(" ");
     }
     System.out.println(s);
   }
}
```

Figura 1: Listagem parcial da aplicação orientada a aspectos para gráficos 2D.

grafo construído para representar o fluxo de controle do método baseado nesse bytecode é ilustrado na Figura 3. Os rótulos dos nós regulares correspondem aos contadores de programa da primeira instrução de bytecode do bloco correspondente. Nós tracejados represen-

```
0:    aload_1
1:    ifnonnull     #8
...
60:   aload_0
61:   getfield      components.LineSegment.p1
      Lcomponents/Point;
64:   invokevirtual components.Point.getX
67:   dstore        %8
69:   aload_0
70:   getfield      components.LineSegment.p1
      Lcomponents/Point;
73:   invokevirtual components.Point.getY
76:   dstore        %10
78:   goto      #200 81 aload_1
82:   invokevirtual #48 <Method double getX()>
85:   aload_0
86:   getfield #17 <Field components.Point p1>
89:   invokevirtual #48 <Method double getX()>
92:   dsub
93:   aload_0
94:   getfield      components.LineSegment.p2
      Lcomponents/Point;
97:   invokevirtual components.Point.getX
100:  aload_0
101:  getfield      components.LineSegment.p1
      Lcomponents/Point;
104:  invokevirtual components.Point.getX
...
132:  invokevirtual components.Point.getY
137:  dadd
138:  dstore        %6
140:  dload     %6
142:  dload     %4
144:  ddiv
145:  dstore_2
146:  aload_0
147:  getfield      components.LineSegment.p1
      Lcomponents/Point;
150:  invokevirtual components.Point.getX
...
200:  new #61 <Class components.FigureFactory>
203:  dup
204:  invokespecial #62 <Method
      FigureFactory()>
207:  astore 12
209:  aload 12
211:  dload 8
213:  dload 10
215:  invokestatic  aspects.FigureLogging.
      aspectOf()Laspects/FigureLogging;
218:  getstatic components.LineSegment.ajc$
      tjp_2 Lorg/aspectj/lang/JoinPoint$
      StaticPart;
221:  invokevirtual
      aspects.FigureLogging.
      ajc$before$aspects_FigureLogging$1$
      114ea88f (Lorg/aspectj/lang/JoinPoint$
      StaticPart;)
224:  invokevirtual Method components.Point
      makePoint(double, double)
...
227:  invokestatic aspects.FigureLogging.
      aspectOf() Laspects/FigureLogging;
230:  invokevirtual
      aspects.FigureLogging.
      ajc$afterReturning$
      aspects_FigureLogging$
      2$114ea88f
233:  nop
...
237:  aload %13
239:  invokestatic  aspects.FigureLogging.
      aspectOf() Laspects/FigureLogging;
242:  getstatic  components.LineSegment.ajc$
      tjp_3 Lorg/aspectj/lang/JoinPoint$
      StaticPart;
245:  invokevirtual
      aspects.FigureLogging.
      ajc$before$aspects_FigureLogging$1$
      114ea88f (Lorg/aspectj/lang/JoinPoint$
      StaticPart;)
248:  invokevirtual
      components.Point.distance
      (Lcomponents/Point;)D (68)
251:  invokestatic
      aspects.FigureLogging.aspectOf()
      Laspects/FigureLogging;
254:  invokevirtual
      aspects.FigureLogging.
      ajc$afterReturning$
      aspects_FigureLogging$
      2$114ea88f
257:  nop
258:  dstore        %14
260:  dload     %14
262:  dreturn
```

Figura 2: Bytecode parcial resultante do método `distance` da classe `LineSegment`.

tam as interceptações e informam, além do contador de programa da primeira instrução de bytecode do bloco, qual sugestão afeta aquele ponto e a qual aspecto pertence (por exemplo $<< before - FigureLogging >>$). Nós em negrito representam os nós de saída e nós formados por círculos duplicados representam chamadas a métodos. Esse tipo de grafo é definido na Seção 4.

3. Teste Estrutural de Programas Orientados a Aspectos

Uma verificação completa de um determinado programa P poderia ser obtida testando P com um conjunto de casos de teste $\mathcal{T}$ que testa todos os elementos do domínio. Entretanto, como geralmente o conjunto de elementos do domínio é infinito ou muito grande, torna-se necessária a obtenção de subconjuntos desses casos de teste. Para isso, podem ser utilizados critérios de teste que auxiliam o testador fornecendo um método para a avaliação de conjuntos de casos de teste e uma base para a seleção de casos de teste. No primeiro caso, os critérios de adequação servem para evidenciar a suficiência da atividade de teste e no segundo caso, para ajudar na

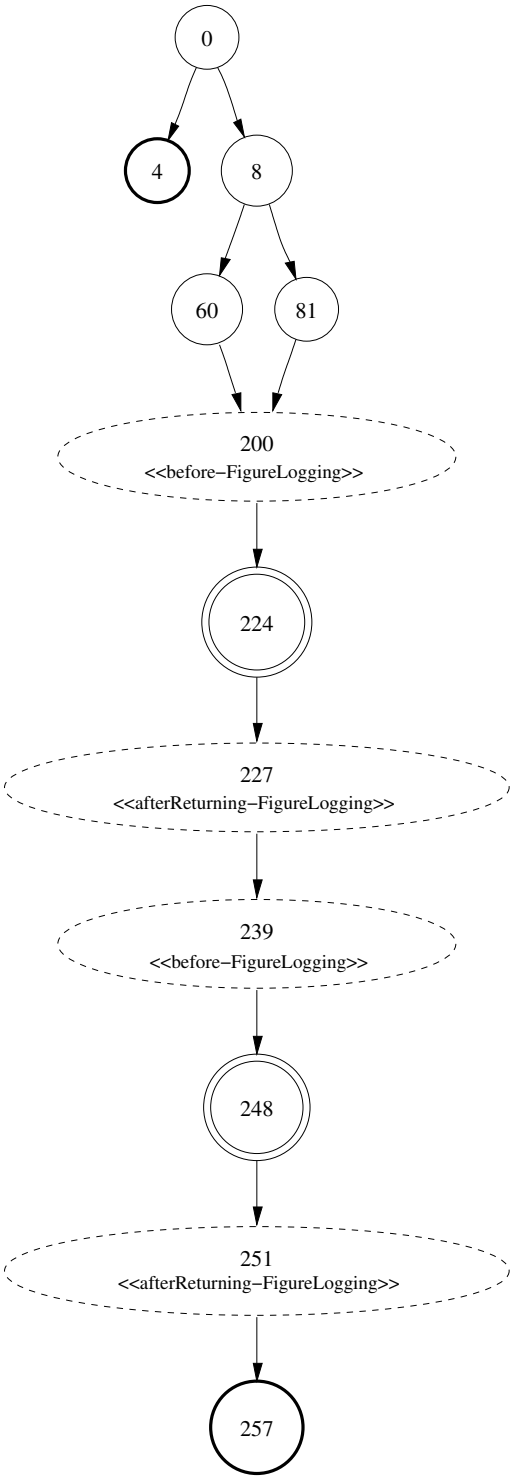

Figura 3: Grafo construído a partir do bytecode resultante do método `distance`.

construção de casos de teste [5].

A técnica de teste estrutural é vista como complementar à técnica funcional e baseia-se na estrutura de um programa para derivar seus casos de teste. Geralmente essa técnica é aplicada ao teste de unidade, apesar de existirem alguns trabalhos relacionados ao teste estrutural de integração [7, 11, 6]. Os critérios da técnica estrutural utilizam o *grafo de fluxo de controle* (GFC) ou *grafo de programa*, que representa o fluxo de controle lógico de um programa utilizando uma notação padrão [13].

Os critérios de teste estruturais são baseados na idéia de que não se pode confiar em uma ati-

vidade de testes se existem certos caminhos que ainda não foram executados no programa P sendo testado. Esses critérios geralmente associam um conjunto $\mathcal{T}$ de casos de teste (que testam um subconjunto das entradas do programa) com um conjunto Π de caminhos no grafo de fluxo de controle de P, que são percorridos quando esses casos de teste são executados. O conjunto de casos de teste $\mathcal{T}$ satisfaz o critério C para P ("$\mathcal{T}$ é C-adequado para o teste de P") se, e somente se, todo caminho requerido por C é um sub-caminho de um dos caminhos de Π [4].
Para se aplicar a técnica de teste estrutural, deve ser definido um modelo de fluxo de controle subjacente a partir do qual podem ser derivados os requisitos de teste para cada critério de teste. Este artigo baseia-se no modelo subjacente adotado pelo AspectJ e considera-se como módulo a ser testado cada método, sugestão ou método introduzido a partir de uma declaração inter-tipo encontrados no programa (todas as construções similares a métodos encontradas no AspectJ). A partir daí, o teste de unidade pode ser feito em cada um desses módulos.
Com a estratégia de implementação adotada pelo AspectJ (Seção 2.1.1), pode-se identificar no bytecode resultante da compilação as interceptações feitas pelos aspectos (que são as chamadas às sugestões). Isto é, cada chamada feita a uma sugestão é na verdade uma interceptação da sugestão de um dado aspecto. Além disso, como as sugestões são implementadas como métodos no bytecode, o mesmo modelo utilizado para a representação do fluxo de controle de métodos é utilizado também para as sugestões e essas podem ser tratadas separadamente. A partir desses resultados, um modelo de fluxo de controle baseado no bytecode pode representar completamente o fluxo de um programa orientado a aspectos de maneira a aplicar a técnica de teste estrutural.

4. Definição de Critérios Estruturais para Programas Orientados a Aspectos

Os modelos e critérios definidos neste artigo são baseados no trabalho de Vincenzi [15], que definiu um modelo e critérios de fluxo de controle para o teste de programas escritos em Java, a partir do bytecode. Na Seção 4.1 o modelo de fluxo de controle definido por Vincenzi [15] é estendido para representar os nós e arestas de interceptação e na Seção 4.2 são definidos os critérios baseados nesse modelo.

4.1. Modelo de Fluxo de Controle

Neste artigo, o grafo de fluxo de controle orientado a aspectos ($\mathcal{AOCFG}$) é o modelo do qual requisitos de teste de fluxo de controle são derivados. Esse grafo é construído para cada unidade, ou seja, cada método e construção similar a método do AspectJ.
Antes da construção do grafo $\mathcal{AOCFG}$, constrói-se o grafo de instruções ($\mathcal{IG}$) de cada módulo. Informalmente, o $\mathcal{IG}$ é um grafo no qual os nós contêm uma única instrução de bytecode e arestas conectam instruções que podem ser executadas seqüencialmente[2]. Para construir os grafos $\mathcal{IG}$ e $\mathcal{AOCFG}$, deve ser feita uma análise estática das instruções de bytecode.
A idéia do grafo $\mathcal{IG}$ é de abstrair o fluxo de controle envolvido em cada instrução de bytecode. Formalmente, o grafo $\mathcal{IG}$ de um módulo m é definido como um grafo dirigido $\mathcal{IG}(m) = (NI, EI, si, TI, II)$, tal que:

- NI representa o conjunto não vazio de nós de um grafo $\mathcal{IG}$: $NI = \{n_i | n_i$ corresponde a uma instrução de bytecode i, para toda instrução de bytecode alcançável i de $m\}$.

[2]Note-se que os resíduos (Seção 2.1.1) inseridos pelo AspectJ encontrados no bytecode também são instrumentados. Futuramente se estudará se esses tipos de instruções devem ter algum tratamento especial (como por exemplo serem destacadas no grafo). Existem também algumas construções adicionadas pelo compilador do AspectJ (que são anotadas com o atributo `AJ_SYNTHETIC` nos arquivos **`.class`** [8]). Supõe-se que essas construções poderiam ser ignoradas na instrumentação, já que servem somente ao mecanismo subjacente do AspectJ.

- EI é o conjunto completo de arestas do $\mathcal{IG}$.
- $si \in NI$ é o nó de entrada, ou seja, ele corresponde ao nó que contém a primeira instrução do módulo m. Seja x um nó de um grafo dirigido, $IN(x)$ corresponde ao número de arestas que chegam em x e $OUT(x)$ corresponde ao número de arestas que saem de x. Tem-se que $IN(si) = 0$.
- $TI \subseteq NI$ é o conjunto de nós de saída, formalmente $TI = \{n_i \in NI | OUT(n_i) = 0\}$.
- $II \subseteq NI$ é o conjunto de nós de interceptação, isto é, nós que representam interceptação de sugestões de aspectos. De fato, esses nós correspondem às instruções que fazem chamadas às sugestões (Seção 2.1.1).

O grafo $\mathcal{IG}$ oferece um modo prático de se percorrer o conjunto de instruções de um módulo m, identificando-se também as interceptações que ocorrem em um programa orientado a aspectos. Entretanto o número de nós e de arestas nesse tipo de grafo podem ser demasiadamente grandes. Dessa maneira é construído o grafo $\mathcal{AOCFG}$ utilizando-se o conceito de *blocos de instrução*, isto é, grupos de instruções que são executadas seqüencialmente em uma execução normal do programa. Quando a primeira instrução do bloco é executada, considera-se que as próximas instruções do bloco sempre são executadas. O grafo $\mathcal{AOCFG}$ é então o modelo base para se derivar requisitos de teste baseados no fluxo de controle para o teste de unidade de programas implementados utilizando o AspectJ. Os rótulos dos nós do $\mathcal{AOCFG}$ são os primeiros contadores de programa das primeiras instruções de bytecode de cada bloco. Os nós de interceptação são rotulados também com o tipo da sugestão e o nome do aspecto ao qual ela pertence.
Um grafo $\mathcal{AOCFG}$ de um dado módulo m é definido como um grafo dirigido $\mathcal{AOCFG}(m) = (N, E, s, I, T)$, tal que cada nó $n \in N$ representa um bloco de instruções:

- N representa o conjunto de nós de um grafo $\mathcal{AOCFG}$: $N = \{n | n$ corresponde a um bloco de instruções de bytecode de $m\}$, ou seja N é um conjunto não vazio de nós, representando todos os blocos de instruções de bytecode de m; I_n é a n-upla ordenada de instruções agrupadas no nó n;
- E é o conjunto completo de arestas do $\mathcal{AOCFG}$. Seja $\mathcal{IG}\ (m) = (NI, EI, si, TI)$, tem-se:
  - E é o conjunto de arestas definido como $E = \{(n_i, n_j)|$ existe em $\mathcal{IG}\ (m)$ uma aresta do último elemento de I_{n_i} para o primeiro elemento de $I_{n_j}\}$;
  - E_i é o conjunto de arestas de interceptação definido como $E_i = \{(n_i, n_j)|$ existe em $\mathcal{IG}\ (m)$ uma aresta do último elemento de I_{n_i} para o primeiro elemento de I_{n_j} e algum elemento de I_{n_j} é uma instrução de interceptação, ou seja, $n_j \in I$, ou existe uma aresta em $\mathcal{IG}\ (m)$ do último elemento de I_{n_i} para o primeiro elemento de I_{n_j} e algum elemento de I_{n_i} é uma instrução de interceptação, ou seja $n_i \in I$ $\}$
- $s \in N \mid IN(s) = 0$ é o nó de entrada de m;
- $I \subseteq N$ é o conjunto (possivelmente vazio) de nós de interceptação. Nesse caso, um nó de interceptação corresponde a um bloco de instruções na qual uma das instruções representa uma interceptação de uma dada sugestão;
- $T \subseteq N$ é o conjunto (possivelmente vazio) de nós de saída, ou seja, $T = \{n \in N \mid OUT(n) = 0\}$;

Os grafos $\mathcal{IG}$ e $\mathcal{AOCFG}$ são estendidos dos modelos definidos por Vincenzi [15]. Na Figura 4 é mostrado o algoritmo utilizado para transformar um grafo de instruções $\mathcal{IG}$ em um grafo de fluxo de controle $\mathcal{AOCFG}$. O algoritmo também é baseado em Vincenzi [15], estendido para representar os nós e arestas de interceptação. Em particular, nas linhas 15 a 20 os nós de interceptação são detectados e adicionados ao conjunto I e garante-se que exista apenas uma interceptação em cada bloco. Além disso, nas linhas 5 a 7, o conjunto de arestas de interceptação E_i é identificado.

```
# Entrada: IG , o grafo de instruções IG =< NI, EI, si, TI, II >
#          a ser reduzido;
# Saída: AOCFG , o grafo de fluxo de controle AOCFG =< N, E, s, T, I >
01  s := NovoBloco(si)
02  para cada x ∈ N
03      se x não tem sucessores
04          T := T ∪ {x}
05  para cada (x, y) ∈ E
06      se y ∈ I ou x ∈ I
07          E_i := E_i ∪ {(x, y)}
# Função auxiliar: NovoBloco
# Entrada: Um nó y de um grafo IG
# Saída: Um bloco do grafo AOCFG
08  ins := a instrução de bytecode em y
09  se y já foi visitado
10      retorne o nó w ∈ N que contém y
11  BlocoAtual := novobloco
12  N := N ∪ {BlocoAtual}
13  x := y
14  faça
15      se x ∈ II //x é um nó de interceptação de IG
16          se BlocoAtual ∈ I //se o bloco atual já tinha uma instrução de interceptação
17              E := E ∪ {(BlocoAtual, NovoBloco(x))}
18              x := null
19          senão
20              I := I ∪ {BlocoAtual}
21      se x ≠ null
22          inclua x como parte do BlocoAtual
23          marque x como visitado
24          se x terminar o bloco atual
25              para cada v tal que (x, v) ∈ EI
26                  E := E ∪ {(BlocoAtual, NovoBloco(v))}
27              x := null
28          senão
29              se existe um v tal que (x, v) ∈ EI
30                  x := v
31              senão x := null
32  enquanto x ≠ null
33  retorne BlocoAtual
```

Figura 4: Algoritmo para gerar um grafo $\mathcal{AOCFG}$ a partir de um grafo $\mathcal{IG}$.

4.2. Critérios de Teste Baseados no Fluxo de Controle

Dois critérios baseados em fluxo de controle – todos-nós e todas-arestas – definidos por Myers [12] foram revisitados para serem aplicados no contexto de programas orientados a aspectos. Além disso são definidos dois outros critérios relacionados com esses mas que são específicos de programas orientados a aspectos, baseados no modelo de fluxo de controle derivado apresentado anteriormente.

Seja $\mathcal{T}$ um conjunto de casos de teste para um programa P ($\mathcal{AOCFG}$ é o grafo de fluxo de controle orientado a aspectos correspondente de P), e seja Π o conjunto de caminhos executados por $\mathcal{T}$. Diz-se que um nó i está incluído em Π se Π contém um caminho $(n_1, \ldots, n_m)$ tal que $i = n_j$ para algum $j, 1 \leq j \leq m$. Similarmente, uma aresta (i_1, i_2) é incluída em Π se Π contém um caminho $(n_1, \ldots, n_m)$ tal que $i_1 = n_j$ e $i_2 = n_{j+1}$ para algum $j, 1 \leq j \leq m - 1$. O grafo $\mathcal{AOCFG}$ mostrado na Figura 3 (Seção 2.1.1) é utilizado para ilustrar cada critério de teste.

4.2.1. O critério todos-nós.

- Π satisfaz o critério todos-nós se cada nó $n \in N$ de um grafo $\mathcal{AOCFG}$ está incluído em Π. Em outras palavras este critério garante que todas as instruções (comandos) de um dado módulo tenham sido executadas ao menos uma vez por algum caso de teste de $\mathcal{T}$.

Outro critério relacionado com o todos-nós mas que é particular de programas orientados a aspectos e o critério todos-nós-de-interceptação:

- todos-nós-de-interceptação

– Π satisfaz o critério todos-nós-de-interceptação se cada nó $n_i \in I$ está incluído em Π. Em outras palavras, este critério garante que cada interceptação ocorrida em um módulo m, e portanto um nó de interceptação do grafo $\mathcal{AOCFG}$ correspondente, seja executada ao menos uma vez por algum caso de teste $\mathcal{T}$.

Na Figura 5 são ilustrados os elementos requeridos (nós preenchidos) para o critério (a) todos-nós e para o critério (b) todos-nós-de-interceptação para o grafo exemplo apresentado anteriormente.Observa-se que os requisitos de testes obtidos a partir do critério todos-nós-de-interceptação são um subconjunto dos requisitos exigidos pelo critério todos-nós.

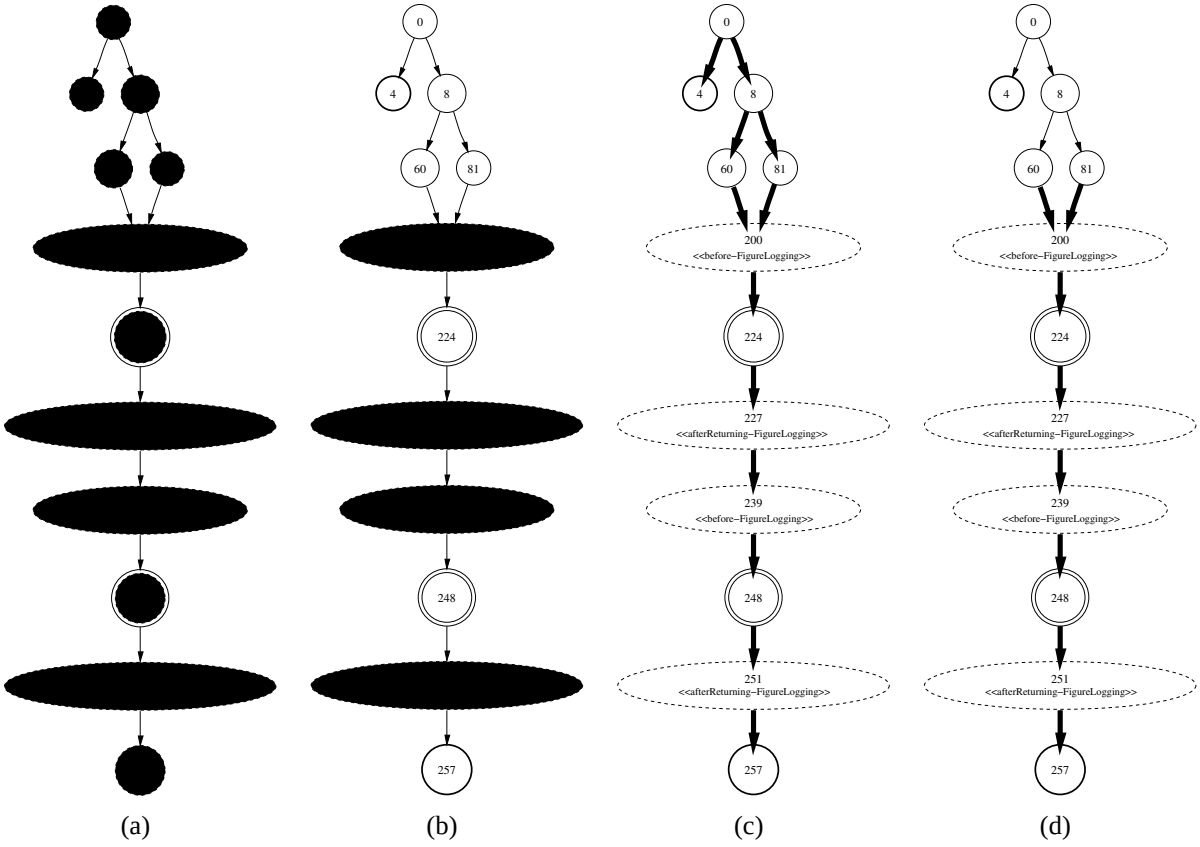

Figura 5: Elementos requeridos para os critérios (a) todos-nós, (b) todos-nós-de-interceptação, (c) todas-arestas e (d) todas-arestas-de-interceptação para o grafo exemplo mostrado na Figura 3.

4.2.2. O critério todas-arestas.

- Π satisfaz o critério todas-arestas se cada aresta $e \in E$ de um grafo $\mathcal{AOCFG}$ está incluída em Π. Em outras palavras, este critério requer que cada aresta de um grafo $\mathcal{AOCFG}$ seja exercitada ao menos uma vez por algum caso de teste de $\mathcal{T}$.

Outro critério relacionado com o todas-arestas mas que é particular de programas orientados a aspectos e ó critério todas-arestas-de-interceptação:

- todas-arestas-de-interceptação
  - Π satisfaz o critério todas-arestas-de-interceptação se cada aresta $e_i \in E_i$ está incluída em Π. Em outras palavras, este critério garante que cada aresta do grafo $\mathcal{AOCFG}$ de um dado módulo cujo nó destino ou fonte é um nó de interceptação seja executada pelo menos uma vez por algum caso de teste $\mathcal{T}$.

Na Figura 5 são ilustrados os elementos requeridos (arestas mais grossas) para os critérios (c) todas-arestas e (d) todas-arestas-de-interceptação para o grafo exemplo apresentado anteriormente. Observa-se que os requisitos de testes obtidos a partir do critério todas-arestas-de-interceptação são um subconjunto dos requisitos exigidos pelo critério todas-arestas.

5. Exemplo de Aplicação da Abordagem

Na Figura 3 foi mostrado o grafo $\mathcal{AOCFG}$ construído a partir do bytecode do método `distance` da classe `LineSegment` (Figura 2 da Seção 2.1.1). O aspecto `FigureLogging` afeta o método `distance` antes e depois das chamadas aos métodos `makePoint` e `distance` (da classe `Point`). Esses pontos de interceptação são identificados no bytecode e representados como nós de interceptação no grafo $\mathcal{AOCFG}$ derivado (nós tracejados).

Baseado no $\mathcal{AOCFG}$ construído, na Tabela 2 são mostrados todos os elementos de teste requeridos para cada um dos critérios previamente definidos (mesmos elementos representados na Figura 5). Com esses elementos coletados, casos de teste podem ser construídos de maneira a exercitar cada um deles, como especificado em um plano de teste. Por exemplo, o testador pode querer cobrir todos-nós-de-interceptação mas não cobrir todos-nós. Para isso ele deve projetar os casos de teste que irão exercitar somente os nós de interceptação, sem se preocupar com os outros nós.

Um exemplo de caso de teste para executar todos os nós de interceptação do método `distance` poderia ser o seguinte: primeiramente constrói-se um segmento de linha composto pelos pontos $P_1 = (0, 1)$ e $P_2 = (0, -1)$. A partir do objeto, o caso de teste consiste em chamar o método `distance` com o ponto $(4, 3)$ como entrada. A saída esperada é o número 4 (a menor distância entre a reta que contém o segmento e o ponto). A partir da execução desse caso de teste, os nós de interceptação localizados antes e depois das chamadas aos métodos `makePoint` (para construir o ponto da reta ortogonal ao ponto dado) e `distance` (que calcula a distância entre pontos), são executados e erros contidos em tais pontos seriam revelados. É interessante perceber que uma das arestas de interceptação não é coberta com a execução desse caso de teste e portanto o critério todas-arestas-de-interceptação não é satisfeito. Para executar a outra aresta de interceptação ($(60, 200)$), deveria ser projetado um caso de teste cujos pontos inicial e final do segmento de linha fossem coincidentes. Dessa maneira a variável `bot` fica com o valor 0 e a aresta que chega à interceptação da pré-sugestão a partir dessa condição é executada. Na Figura 6 são mostrados esses dois casos de teste escritos utilizando o framework de teste JUnit [2]. O objeto `ff` da classe `FigureFactory` é utilizado para a criação de figuras. Cada método iniciado em "test" representa um caso de teste.

No caso da pré-sugestão do aspecto `FigureLogging` (cujo grafo $\mathcal{AOCFG}$ está representado na Figura 7), para o critério todos-nós, necessitar-se-ia da criação de casos de teste que chamassem algum método interceptado das classes `Point`, `LineSegment` e de alguma outra classe do pacote `components`. Dessa maneira, cada uma das condições seria satisfeita e todos os nós seriam cobertos.

O teste das sugestões isoladas necessita ainda de algumas considerações. No desenvolvimento de programas orientados a aspectos, os aspectos podem ser genéricos (isto é, podem ser construídos para serem utilizados em qualquer aplicação) ou específicos de uma dada aplicação. Para testar aspectos genéricos que ainda não foram integrados a uma aplicação, para os casos de teste é necessária a criação de uma ou mais classes que sejam interceptadas de maneira a sensibilizar os aspectos. Por exemplo, dado um aspecto genérico de registro de execução (*trace*), para se testar os métodos e sugestões envolvidos é necessário criar uma ou mais classes que sejam interceptadas pelas sugestões do aspecto e executar os métodos interceptados para que as sugestões e os métodos chamados por elas sejam executados. Já no caso dos aspectos específicos, é necessário apenas executar as partes dos componentes que sensibilizam os aspectos para que a análise de cobertura seja feita. O aspecto `FigureLogging` apresentado neste artigo é um exemplo de aspecto específico da aplicação e, sendo assim, para se executar as sugestões e métodos envolvidos é necessário somente chamar os métodos que são interceptados pelo aspecto.

```
import junit.framework.*;
public class GeometryTestCase extends TestCase {
    private FigureFactory ff = new FigureFactory();
    public GeometryTestCase( String str ) {
        super( str );
    }
    public GeometryTestCase(  ) {
        this( "" );
    }
    //covers all interception nodes
    public void testCoverLineSegmentDistanceInterceptions() {
        Point p1 = ff.makePoint(0, 1);
        Point p2 = ff.makePoint(0, -1);
        Point p3 = ff.makePoint(4, 3);
        LineSegment l = ff.makeLineSegment(p1, p2);
        double d = l.distance(p3);
        assertEquals(4, d, .0);
    }

    //covers the other interception edge which was not covered
    //by the previous test case
    public void testCoverLineSegmentInterceptionEdges0Length() {
        Point p1 = ff.makePoint(0, 0);
        Point p2 = ff.makePoint(0, 0);
        Point p3 = ff.makePoint(4, 3);
        LineSegment l1 = ff.makeLineSegment(p1, p2);
        double d = l1.distance(p3);
        assertEquals(5, d, .0);
    }
}
```

Figura 6: Casos de teste para cobrir os nós e arestas de interceptação do método `distance` da classe `LineSegment`.

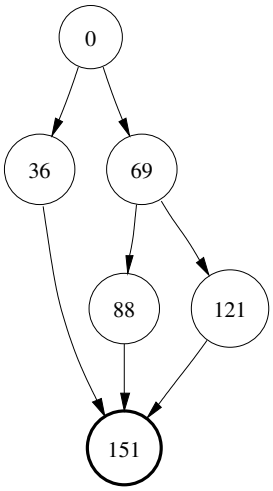

Figura 7: Grafo $\mathcal{AOCFG}$ da pré-sugestão do aspecto `FigureLogging`.

O exemplo apresentado não contém nenhuma sugestão substitutiva (`around advice`) e nenhum método introduzido, porém testes já foram feitos com esses tipos de construções. No caso das sugestões substitutivas, o compilador do AspectJ substitui o ponto de junção por um bloco que chama a sugestão substitutiva e, dessa maneira, os nós de interceptação para as sugestões substitutivas também são representados. Apesar disso, quando se utiliza o método `proceed` (que faz com que o ponto de junção original seja executado) dentro da sugestão, um outro método é criado dentro da classe (com o nome de `NomeDoMétodo_aroundBodyN`) contendo o ponto de junção original. Sendo assim, elementos de teste requeridos baseados nos critérios podem ser coletados tanto para o ponto de junção original – encapsulado nos métodos `NomeDoMétodo_aroundBodyN` – quanto para a sugestão substitutiva. Quanto aos métodos introduzidos, esses são tratados como se já existissem na classe e elementos de teste requeridos também são coletados normalmente.

Tabela 2: Conjuntos de elementos requeridos para o método `distance`.

Critério	**Elementos Requeridos**
todos-nós-de-interceptação	$\{200, 227, 239, 251\}$
todas-arestas-de-interceptação	$\{(60, 200), (81, 200), (200, 224), (224, 227), (227, 239), (239, 248), (248, 251), (251, 257)\}$
todos-nós	$\{0, 4, 8, 60, 81, 200, 224, 227, 239, 248, 251, 257\}$
todas-arestas	$\{(0, 4), (0, 8), (8, 60), (8, 81), (60, 200), (81, 200), (200, 224), (224, 227), (227, 239), (239, 248), (248, 251), (251, 257)\}$

Na Tabela 3, para se ter uma idéia do esforço envolvido, são listados os números de casos de teste necessários para a cobertura de cada classe da aplicação parcialmente listada na Seção 2.

Tabela 3: Número de casos de teste necessários para cada classe e para cada critério da aplicação orientada a aspectos.

Classe/Aspecto	**Critério**	**Nº de C.T.**
`Point`	todos−nós	5
	todos−nós−de−interceptação	5
	todas−arestas	5
	todas−arestas−de−interceptação	5
`LineSegment`	todos−nós	10
	todos−nós−de−interceptação	5
	todas−arestas	10
	todas−arestas−de−interceptação	6
`FigureLogging`	todos−nós	3
	todos−nós−de−interceptação	0
	todas−arestas	3
	todas−arestas−de−interceptação	0
`PointBoundsChecking`	todos−nós	15
	todos−nós−de−interceptação	0
	todas−arestas	15
	todas−arestas−de−interceptação	0

6. Conclusão e Trabalhos Futuros

Este trabalho representa um passo inicial na aplicação do teste estrutural em programas orientados a aspectos. A intenção é de se avaliar o uso e efetividade dos critérios na prática por meio de experimentação, e futuramente estender os modelos para definição de outros critérios. Supõe-se, por exemplo, que a extensão do modelo apresentado neste artigo para o teste de fluxo de dados de integração pode ser útil, já que se consideraria acessos a dados (que permitem a definição de critérios mais fortes) nas interações entre aspectos e componentes.

Em comparação com o trabalho de Zhao, de maneira geral, o modelo proposto neste artigo pode ser considerado mais direto e simplificado. O problema da proposta de Zhao é que os aspectos não podem ser considerados separadamente dos componentes. Os módulos propostos (*e.g. clustering-methods*, *clustering-advices* [17, 18]) misturam as sugestões com os componentes que elas interceptam e dessa maneira nem as classes nem os aspectos podem ser analisados por si próprios. Isto pode se tornar um problema quando existem vários aspectos afetando várias classes. Supõe-se que tal abordagem tenha sido inspirada pela estratégia de implementação

adotada pelo AspectJ nas versões anteriores à 1.1.1 mas não é explicitamente mostrado como os modelos são derivados. Além disso, não é considerado que aspectos também podem afetar os próprios aspectos e nenhum critério de teste é definido.
A ferramenta de teste JaBUTi [16], desenvolvida originalmente para o teste de programas Java, será estendida para o teste de programas orientados a aspectos. No presente momento testes de unidade já podem ser feitos para os critérios todos-nós e todas-arestas (além do critério de fluxo de dados todos-usos). Apesar disso, nenhuma informação especial dos programas orientados a aspectos é apresentada.
O modelo e critérios definidos neste artigo também serão estendidos para contemplar o tratamento de exceções de Java, seguindo o trabalho de Vincenzi [15] (a ferramenta JaBUTi já implementa tal funcionalidade).

Referências

1. AOSD Steering Committee Aspect-Oriented Software Development Community & Conference. Disponível em: `http://www.aosd.net` (Acessado em 14/07/2004)
2. Beck, K.; Gamma, E. JUnit cookbook. Disponível em: `www.junit.org` (Acessado em 14/07/2004)
3. Elrad, T.; Filman, R. E.; Bader, A. Aspect-oriented programming: Introduction. Commun. ACM, v. 44, n. 10, p. 29–32, 2001.
4. Frankl, P. G.; Weyuker, E. J. An Applicable Family of Data Flow Testing Criteria. In: IEEE Transactions on Software Engineering, 1988, p. 1483–1498.
5. Frankl, P. G.; Weyuker, E. J. Testing software to detect and reduce risk. J. Syst. Softw., v. 53, n. 3, p. 275–286, 2000.
6. Haley, A.; Zweben, S. Development and application of a white box approach to integration testing. J. Syst. Softw., v. 4, n. 4, p. 309–315, 1984.
7. Harrold, M. J.; Rothermel, G. Performing Dataflow Testing on Classes. In: Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering, New Orleans, LA, USA: ACM, 1994, p. 154–163.
8. Hilsdale, E.; Hugunin, J. Advice weaving in aspectj. In: Proceedings of the 3rd international conference on Aspect-oriented software development, ACM Press, 2004, p. 26–35.
9. Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold, W. G. An Overview of AspectJ. Lecture Notes in Computer Science, v. 2072, p. 327–355, 2001.
10. Laddad, R. AspectJ in Action – Practical Aspect-Oriented Programming. Greenwich, Connecticut: Manning Publications Co., 2003.
11. Linnenkugel, U.; Müllerburg, M. Test data selection criteria for (software) integration testing. In: Proceedings of the first international conference on systems integration on Systems integration '90, IEEE Press, 1990, p. 709–717.
12. Myers, G. J. The Art of Software Testing. John Wiley and Sons, Inc., 1979.
13. Pressman, R. S. Software Engineering: A Practitioner's Approach. McGraw-Hill Higher Education, 2001.
14. The AspectJ Team Aspectj programming guide. Disponível em: `http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/doc/progguide/index.html` (Acessado em 31/03/2004)
15. Vincenzi, A. M. R. Orientação a objeto: Definição, implementação e análise de recursos de teste e validação. Tese de Doutoramento, Universidade de São Paulo, 2004.
16. Vincenzi, A. M. R.; Wong, W. E.; Delamaro, M. E.; Maldonado, J. C. JaBUTi: A Coverage Analysis Tool for Java Programs. In: 17º Simpósio Brasileiro de Engenharia de Software, Manaus, AM, Brasil, 2003.

17. Zhao, J. Tool Support for Unit Testing of Aspect-Oriented Software. In: OOPSLA 2002 Workshop on Tools for Aspect-Oriented Software Development, Seattle, WA, 2002.
18. Zhao, J. Data-flow-based unit testing of aspect-oriented programs. In: Proceedings of the 27th Annual International Conference on Computer Software and Applications, IEEE Computer Society, 2003, p. 188.