Integration Testing of Aspect-Oriented Programs: a Structural Pointcut-Based Approach*

Otávio Augusto Lazzarini Lemos and Paulo Cesar Masiero

Departamento de Sistemas de Computação, ICMC/USP - São Carlos
Caixa Postal 668 13560-970 São Carlos-SP-Brasil

{oall;masiero}@icmc.usp.br

Resumo. *Várias abordagens de teste focam o descobrimento de falhas em implementações de unidades de software. Um problema não tratado pelo teste de unidade é a interação entre unidades, no que diz respeito à corretude das interfaces. Em Programação Orientada a Aspectos, esse problema é dificultado por mecanismos de conjuntos de junção, que definem interfaces implícitas no programa base. Neste trabalho é apresentada uma abordagem de teste de integração estrutural baseada no mecanismo de conjuntos de junção para programas AspectJ. Um modelo chamado $\mathcal{PCCFG}$ (*Pointcut-based Control Flow Graph*) é definido para representar regiões de execução afetadas por conjuntos de junção. Baseado nesse modelo, dois critérios de fluxo de controle para uma medida de cobertura transversal são propostos: todos-nós-de-adendo e todas-arestas-de-adendo. Como avaliação preliminar da aplicabilidade e efetividade da abordagem proposta, os critérios são implementados em uma ferramenta de teste chamada JaBUTi/PC-AJ e é apresentado um exemplo de aplicação. O exemplo mostra evidências da efetividade dos critérios quando comparados com critérios de teste de unidade.*

Abstract. *Several testing approaches focus on finding faults in software units of implementation (i.e., unit testing). A problem not addressed by unit testing is the interaction among units, with respect to the correctness of their interfaces. With the use of Aspect-Oriented Programming this problem is further complicated by pointcut mechanisms that cut new interfaces in the base program. In this paper a structural integration testing approach for AspectJ programs is presented. A model called $\mathcal{PCCFG}$ (Pointcut-based Control Flow Graph) to represent the flow of control between base units and pieces of advice is defined. Based on the $\mathcal{PCCFG}$, two control-flow criteria for a crosscutting coverage measure are defined: all-pointcut-based-advice-nodes and all-pointcut-based-advice-edges. As a preliminary evaluation of the feasibility and effectiveness of the proposed approach, an implementation of the criteria in an AspectJ testing tool (JaBUTi/PC-AJ) is presented along with an application example. The example shows evidence of the effectiveness of the pointcut-based criteria to find AO related faults compared to unit testing criteria.*

1. Introduction

Aspect-Oriented Programming (AOP) is a technology that builds on top of other software engineering paradigms to support the modularization of crosscutting concerns. To en-

*The authors are financially supported by FAPESP and CNPq.

hance the confidence in AO programs, some researchers have proposed the adaptation of traditional testing techniques to this type of software. However, none of them focus on the pointcut mechanism of AOP to enhance the confidence that the crosscutting concerns correctly interact with base units. In this paper we present an integration testing approach for AO programs consisting of a testing model and criteria based on the pointcut mechanism. These criteria support a measure of adequacy for test sets with respect to advice interactions.

Although AOP seems to partially solve the problem of tangling and scattering concerns, it does not guarantee quality software in terms of correctness. In fact, some authors believe that the AOP power of expressiveness can even lead to new types of faults [Mortensen and Alexander 2005]. Previous work on testing AO programs have targeted some of these shortcomings, but do not focus on the pointcut mechanism [Zhao 2003, Lemos et al. 2007, Franchin 2007].

Pointcut descriptors cut new interfaces in the base structure of a system [Kiczales and Mezini 2005], making pieces of advice be executed at several (join) points. Since some faults might be revealed only when a piece of advice is executed at a particular join point, we propose a coverage measure that ensures the execution of all statements and branches of each advice at each related join point[1]. To support such type of coverage measure, we define a model to represent the execution regions of a program that are affected by pieces of advice through pointcuts. Based on this model, we define two control-flow criteria: the all-pointcut-based-advice-nodes and the all-pointcut-based-advice-edges. We also extend the JaBUTi family of tools [Vincenzi et al. 2006] to support these model and criteria for AspectJ programs. As a preliminary effectiveness evaluation of our approach, we apply the criteria to an AspectJ example.

The example shows evidence that the pointcut-based approach is effective in finding AO specific faults, when compared with unit testing criteria. A preliminary cost evaluation of a previous work on which the approach presented in this paper is related [Franchin et al. 2007], shows that the cost of application of the criteria is relatively low [Lemos et al. 2008]. These two sets of results summed with the extension of the JaBUTi tool, which shows evidence of the feasibility of the approach, represent a preliminary evaluation that motivates the application of our model and criteria. The remainder of this paper is structured as follows. Section 2 presents background on the main topics of this paper: AOP and Software Testing; and Section 3 presents the pointcut-based integration testing approach. Section 4 presents an example of application of the approach and Section 5 shows implementation details of our approach. Finally, Section 6 presents related work and Section 7 concludes the paper with some remarks and future directions.

2. Background

The underlying concept behind Aspect-Oriented Programming (AOP) is that while traditional programming techniques help modularizing different concerns present in a software system, there are still some concerns that cannot be clearly mapped to isolated modules of implementation [Kiczales et al. 1997]. For instance, concerns such as persistence, access control, synchronization policies, and logging; tend to be tangled with and scattered

[1] In a previous paper we presented a preliminary exploration of this idea [Lemos et al. 2006].

throughout the basic modules of implementation (also denominated *base* code). These concerns are often called *crosscutting* concerns [Elrad et al. 2001].

AOP supports the implementation of separate modules – called aspects – that have the ability to cut across other modules, adding behavior that would otherwise be spread throughout the base code. General-purpose AOP languages must define four features: (1) a join point model that describes hooks in the program where additional behavior may be defined; (2) a mechanism for identifying these join points; (3) modules that encapsulate both join point specifications and behavior enhancements; and (4) a *weaving* process to combine both base code and aspects [Elrad et al. 2001].

AspectJ [Kiczales et al. 1997] is an extension of the Java language to support general-purpose AOP. In AspectJ, aspects are modules that combine join point specifications (pointcuts or, more precisely, pointcut designators – PCD[2]); pieces of advice, which implement the desired behavior to be added at join points; and regular OO structures such as methods, fields, and inner classes. Aspects can also declare members (fields and methods) to be owned by other types, *i.e.*, inter-type declarations. AspectJ also supports declarations of warnings and errors that arise when certain join points are identified or reached. Advice can be executed before, after, or around join points selected by the corresponding pointcut, and are implemented as method-like constructs. Advice can also pick context information from the join point that caused them to execute.

2.1. Example

Consider an online system (named Online Music) for playing and displaying information about songs in a large music database. The basic requirements of the system are the following. Each user has an account with a certain balance and can access songs available in the database. At a specific price, stipulated per song, users can play songs, read lyrics, and display related songs. Users are charged the same price for all operations (playing, reading lyrics, and displaying related songs). Users can also display the names of the composers of a song, free of charge. Finally, a song can be a bonus, which makes any operation on it be free of charge (playing, reading lyrics, etc.).

Figure 1 shows a partial class/aspect diagram with the basic classes and aspect of the Online Music system. The main modules are the `Song` class and the `Billing` aspect. The `Song` class represents songs available in the database and has methods responsible for playing, showing lyrics, showing related songs, and showing the names of the composers of a song. The `Billing` aspect is responsible for billing users of the system when using chargeable operations. The `bill` advice inside `Billing` checks whether the user has sufficient balance for executing the related operation and bills him/her according to the stipulated price. If the user does not have sufficient balance, an exception is thrown. `bill` is executed *after returning* from join points defined by the `useTitle` pointcut designator: namely, the execution of the `play` method and the execution of the methods whose names start with "show" (*e.g.*, `showRelated`) in the `Song` class. The partial source code of `Song` and `Billing` are presented in Figures 2 and 3.

[2]A pointcut is the set of selected join points itself and the PCD is usually a language construct that defines pointcuts. For simplicity, we use these terms interchangeably, meaning the language construct that defines a set of join points.

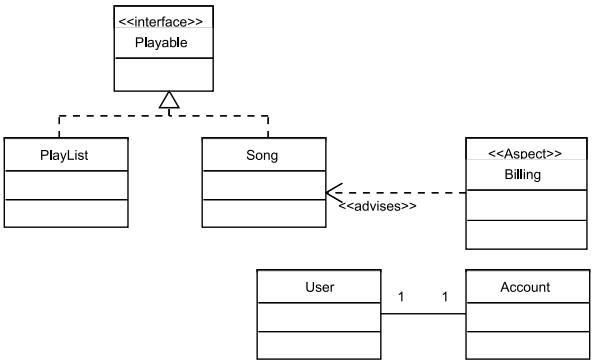


Figure 1. Simplified class diagram of the online music system.

2.2. Testing aspect-oriented programs

Software testing is usually performed at three levels:

1. Unit testing, where the smallest parts of a system are tested in isolation;
2. Integration testing, where interactions among units are tested (*i.e.*, their interfaces); and
3. System testing, which consists in verifying the integration of the general elements of a system to assure that they combine adequately and that expected global functioning/performance is obtained.

In this paper we focus on structural integration testing, building on top of unit and integration testing approaches described before [Lemos et al. 2007, Vincenzi et al. 2005, Vincenzi et al. 2006, Franchin 2007]. We consider a method and an advice as the smallest units to be tested (*i.e.* the *units* targeted by unit testing). In structural testing, the control-flow graph (CFG) is used to represent the flow of control of a program, where nodes represent a statement or a block of statements executed sequentially, and edges represent the flow of control from one statement or block of statements to another [Rapps and Weyuker 1985].

Lemos et al. [Lemos et al. 2007] defined a basic unit testing model for OO and AO Java programs – the aspect-oriented def-use ($\mathcal{AODU}$) graph – which builds on top of the work of Vincenzi et al. (for OO programs only). The $\mathcal{AODU}$ is generated for each unit to be tested, both methods and pieces of advice. It is defined as a directed graph with elements (N, E, s, T, C). Informally, N represents the set of nodes – which are composed by blocks of bytecode instructions that are executed sequentially; E represents the set of edges connecting nodes when there is transfer of flow from one to the other; s represents the entry node; T is the set of exit nodes; and C is the set of nodes affected by pieces of advice (called *crosscutting nodes*). Following Vincenzi et al.'s work, we also differentiate regular edges – edges that connect regular nodes – from exceptional edges – edges that connect regular nodes to nodes that represent exception handling statements. The element C was added to the original def-use model defined by Vincenzi et al. [Vincenzi et al. 2005, Vincenzi et al. 2006] to represent the basic interaction that occurs in AO programs.

The $\mathcal{AODU}$ graph is represented by the following conventions: single circled nodes represent regular blocks of instructions, double circled nodes represent method

```
public class Song implements Playable {
  private String name;
  private boolean bonus;
  private String composer;
  private ArrayList<Song> related = new ArrayList<Song>();

  public Song(String name, String composer, boolean bonus) {
    super();
    this.name = name;
    this.composer = composer;
    this.bonus = bonus;
  }

    ...

  public void play() {
    ...
  }

  public void showLyrics(){
    ...
  }

  public void showRelated() {
    if (!related.isEmpty()) {
      System.out.println("Related songs:");
      for (Iterator<Song> it = related.iterator (); it.hasNext ();) {
        System.out.println(((Song)it.next()).getName());
      }
    }
  }

  public void showComposer(){
    System.out.println("The composer of the song is " + getComposer());
  }

  public boolean equals(Object o){
      Song other = (Song) o;
      return this.name.equals(other.name);
  }

  public int hashCode() {
    return name.hashCode();
  }

  public boolean isBonus() {
    return bonus;
  }

  ...
}
```

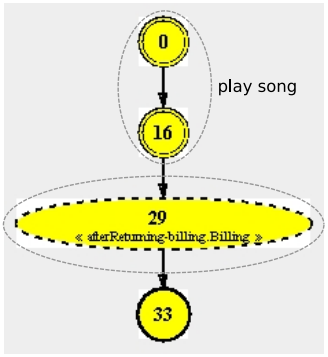
Figure 2. Source code of the Song class.

calls, bold nodes represent exit nodes, dashed elipses (crosscutting nodes) represent advice execution and contain additional information of what kind of advice is affecting that point and to which aspect it belongs, regular edges represent regular control flow, and dashed edges represent exceptional control flow.

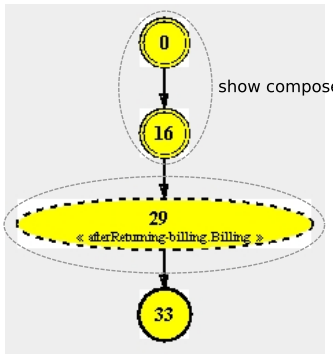
Five examples of $\mathcal{AODU}$ graphs with explanations of each part of the graph are presented in Figures 4 and 5. The units refer to four methods of the `Song` class that are affected by the `bill` advice of the `Billing` aspect, and the advice itself. The code of the two modules were presented in Figures 2 and 3.

```
public aspect Billing {
  public pointcut useTitle() :
        execution(* Song.play(..)) ||
        execution(* Song.show*(..));

  @AdviceName("bill")
  after(Song song) returning throws InsufficientBalanceException :
    useTitle() && this(song) {

    if (song.isBonus())
      return;

    User user = (User)Session.instance().getValue("currentUser");
    int amount = song.getPrice();
    if(amount > user.getAccount().getBalance())
      throw new InsufficientBalanceException(
        "Insufficient available balance.");
    user.getAccount().debit(amount);
    System.out.println("Charge: " + user + " " + amount);
  }
}
```

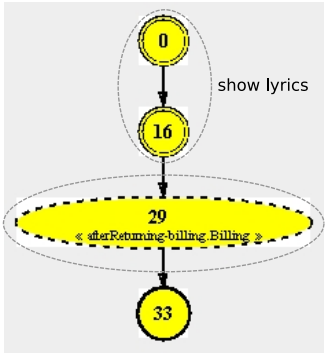
Figure 3. Source code of the `Billing` aspect.



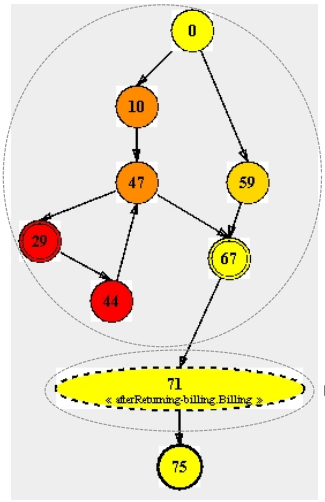
(a) $\mathcal{AODU}$ of the `play` method



(b) $\mathcal{AODU}$ of the `showComposer` method



(c) $\mathcal{AODU}$ of the `showLyrics` method



(d) $\mathcal{AODU}$ of the `showRelated` method

Figure 4. Examples of $\mathcal{AODU}$s.

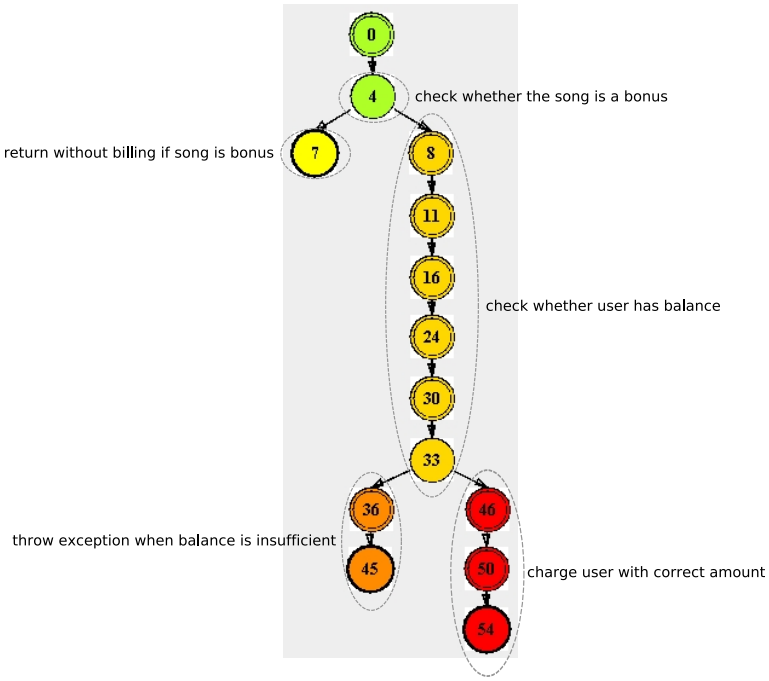


Figure 5. $\mathcal{AODU}$ of the `bill` advice.

2.2.1. Pairwise Structural Integration Testing of AO Programs

Franchin et al. [Franchin et al. 2007] presented a pairwise integration testing approach for Object-Oriented (OO) and AO Java programs that builds on top of the unit testing approach referred to in the last section. A model to represent pairs of interacting units (called Pair-Wise Def-Use Graph – $\mathcal{PWDU}$) was proposed, and a family of structural testing criteria was defined based on such model. The idea is to force the tester to adequately cover the structure of each unit in the contexts that it is integrated. The motivation is that there can be faults in units that are only sensitized in the context of other particular units. Although pairs that may contain an advice as one of the units are treated, no particular attention is given to them (*i.e.* they are treated just as method pairs).

3. Pointcut-based integration testing of aspects

To test aspect-oriented programs, an interesting measure of adequacy for test sets is the global coverage of each crosscutting concern present in the program. For instance, to have more confidence that an access control concern has been thoroughly tested in a program P, it is important to know the coverage of the structure of the corresponding advice at each join point. Based on the pairwise approach discussed before [Franchin et al. 2007], we propose the representation of all pairs of interacting units related to each advice in a single model. The idea is to help understanding and testing the pieces of advice at each join point of the program.

3.1. The $\mathcal{PCCFG}$ graph

To define structural testing criteria based on pointcuts, we need a model to represent the flow of control at join points. We define a graph that must be constructed for each advice-pointcut pair called Pointcut-based Control Flow Graph ($\mathcal{PCCFG}$). The $\mathcal{PCCFG}$

comprises nodes and edges of base units selected by a pointcut and nodes and edges of the corresponding advice (*i.e.*, the $\mathcal{AODU}$s of the base units and the $\mathcal{AODU}$ of the advice) repeated at each joint point. The $\mathcal{PCCFG}$ models execution regions of the program that are affected by a pointcut.

The nodes of the $\mathcal{PCCFG}$ are labeled with two strings: a prefix and a suffix. The suffix always corresponds to the offset of the first bytecode instruction of the corresponding block, such as the labels of the $\mathcal{AODU}$ nodes. The prefix is (1) a capital letter, if the node belongs to a base unit; or (2) the number of the join point, if the node corresponds to the advice structure. A letter is given for each base unit at each join point, and a number is given for each join point. The prefix is used to avoid repetition, since bytecode offsets may repeat from unit to unit. A colon is used to separate the suffix from the prefix. Since we are interested in the coverage of the structure of the advice at each join point, the representation of the base units can be reduced for simplification. Also note that if there are multiple join points in a single unit, the unit will appear multiple times in the graph. An example of a $\mathcal{PCCFG}$ is presented in Figure 6. The graph represents the `bill` advice and corresponding pointcut of the `Billing` aspect presented before, integrated with all the affected join points. Note that the advice affects four join points in the following methods of the `Song` class: `play`, `showLyrics`, `showComposer`, and `showRelated`. The nodes suffixed with 45 do not have edges connecting to the base units' nodes because they represent the throwing of an exception.

3.2. Pointcut-based integration testing criteria

Testing criteria are important to provide systematic selection and evaluation of test sets. To enhance the confidence that each crosscutting behavior implemented as an advice is integrated to a program in a correct way, we propose two control-flow based structural testing criteria. The idea is to make sure that the advice is thoroughly covered at each join point selected by the corresponding pointcut.

Let T be a test set for a program P, being $\mathcal{PCCFG}$ the graph of a set of pairs of units, and let Π be the set of paths executed by T in P. A node $i \in N$ is included in Π if Π contains a path $(n_1, \ldots, n_m)$ where $i = n_j$ for some j, $1 \leq j \leq m$. Similarly, an edge (i_1, i_2) is included in Π if Π contains a path $(n_1, \ldots, n_m)$ where $i_1 = n_j$ and $i_2 = n_{j+1}$ for some j, $1 \leq j \leq m - 1$. Remember that advice nodes are prefixed with numbers (the join point number) in the $\mathcal{PCCFG}$. We define two control-flow criteria based on the traditional *all-nodes* and *all-edges*:

- **all-pointcut-based-advice-nodes** (all-pc-nodes): Π satisfies the all-pc-nodes criterion if each advice node, for each join point selected by the corresponding pointcut, is included in Π. In other words, this criterion requires that each $\mathcal{PCCFG}$ node whose label is prefixed by a number (*i.e.*,the join point number) be exercised at least once by a test case in T. It requires that every statement in the advice be executed at each join point it may run.
- **all-pointcut-based-advice-edges** (all-pc-edges): Π satisfies the all-pc-edges criterion if each advice edge, for each join point selected by the corresponding pointcut, is included in Π. In other words, this criterion requires that each $\mathcal{PCCFG}$ edge whose target and source nodes have labels prefixed by a number (*i.e.*, the join point number) be exercised at least once by a test case in T. It requires that every possible branch in the advice be executed at each join point it may run.

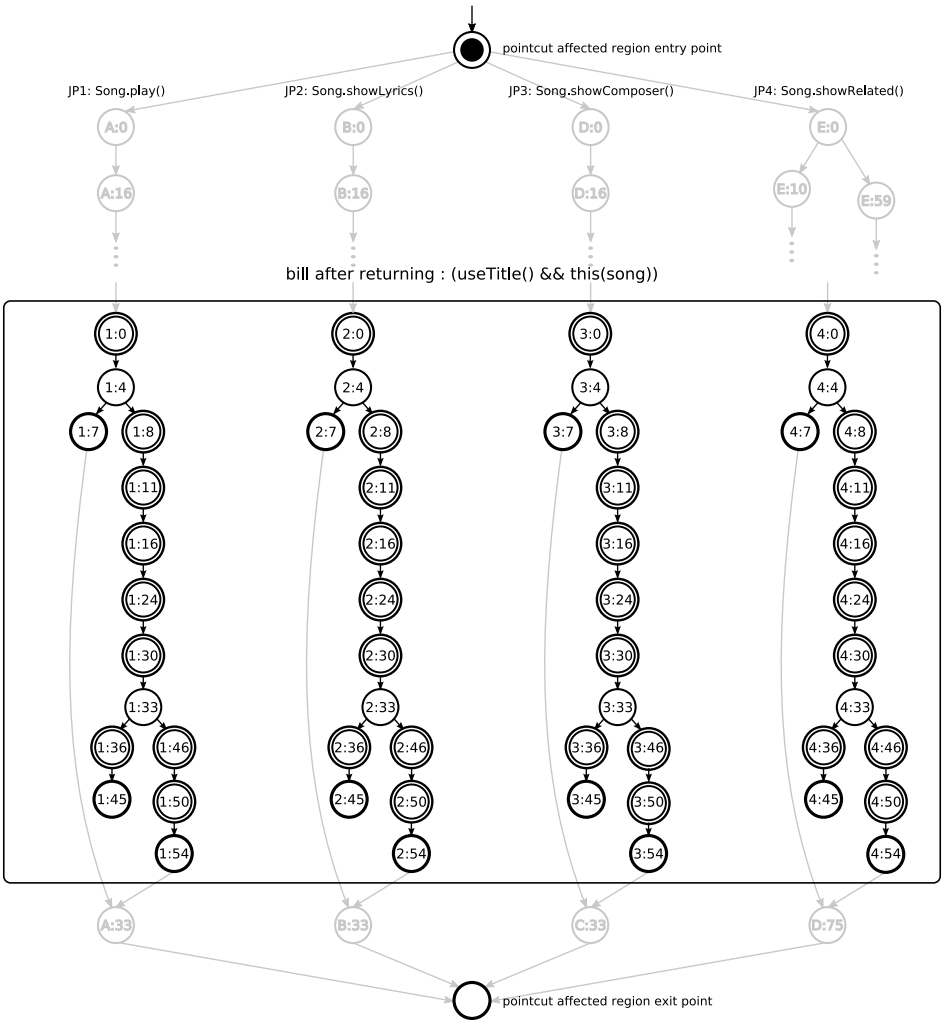


Figure 6. $\mathcal{PCCFG}$ for the `bill` advice and `useTitle` pointcut.

3.3. Basic testing strategy

As pointed out in the beginning of this section, most testing processes divide the testing activity in three levels: (1) Unit testing, (2) Integration testing, and (3) System testing [Bertolino 2007]. Following this strategy, our pointcut-based testing criteria would be more effectively applied after unit testing the program. Thus, the natural testing strategy to be followed in this context would be: (1) focus on each unit by testing each method and advice in isolation (by using, for instance, the criteria proposed before [Lemos et al. 2007, Vincenzi et al. 2005, Vincenzi et al. 2006]); (2) focus on the crosscutting concerns by testing each advice at each selected join point. Afterwards, to have more confidence in the program and to consider the integration of methods, the pairwise testing criteria proposed by Franchin et al. [Franchin et al. 2007] could also be used. Note that, to have a more efficient testing activity, the test set used in the precedent level can be used as a starting point for the next level.

4. Application example

To demonstrate the use of the pointcut-based testing criteria and an evidence of its effectiveness, we will use the Music Online application presented in Section 2. Following

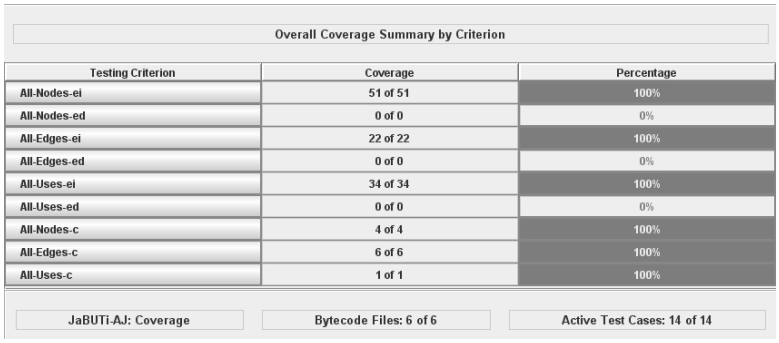
the basic testing strategy presented in the last section, we should first unit test each of the methods and advice present in the program. For that purpose, we can use a functional testing approach and check the structural coverage by using the criteria proposed by Vincenzi et al. [Vincenzi et al. 2005, Vincenzi et al. 2006] and Lemos et al. [Lemos et al. 2007]. Figure 7 shows part of a complete functional test set implemented in JUnit for the Music Online application adequate for the structural unit testing criteria. We can see that the test set is adequate by using the JaBUTi/AJ tool [Lemos et al. 2007] (see Figure 8). Even though the test set is adequate for these criteria, no faults are revealed when they execute.

```
public class InitialTS extends TestCase {
  static final Song TEST_SONG1 =
    new Song("test song1", "Bach", 10, false);
  static final Song TEST_SONG2 =
    new Song("test song2", "Beethoven", 10, true);
  static final Song TEST_SONG3 =
    new Song("test song3", "Dave Brubeck", 10, false);
  ...

  public void testEquals() {
    ...
    assertTrue(s1.equals(TEST_SONG1));
  }

  public void testPlaySong() {
    ...
    assertTrue(strOut.indexOf
      ("Playing song test song1") != -1);
  }

  public void testShowLyrics() {
    ...
    assertTrue(strOut.indexOf
      ("Displaying lyrics for test song1") != -1);
  }

  public void testShowComposer() {
    ...
    TEST_SONG2.showComposer();
    String strOut = baos.toString().trim();
    assertTrue(strOut.indexOf("Beethoven") != -1);
  }

  public void testSongHashcode() { ... }

  public void testDebit() {
    ...
    user.getAccount().credit(10);
    TEST_SONG1.play();
    assertEquals(0, user.getAccount().getBalance());
  }

  public void testInsufficientBalance() {
    ...
    user.getAccount().credit(9);
    try {
      TEST_SONG1.play();
      fail();
    } catch (InsufficientBalanceException e) { }
  }

  public void testDontChargeBonus() {
    ...
    user.getAccount().credit(9);
    TEST_SONG2.play();
    assertEquals(9, user.getAccount().getBalance());
  }

  public void testDontChargeShowComposer() {
    ...
90  user.getAccount().credit(9);
91  TEST_SONG2.showComposer();
92  assertEquals(9, user.getAccount().getBalance());
  }

  public void testRelated() {
    ...
    TEST_SONG2.showRelated();
    String strOut = baos.toString().trim();
    assertTrue(strOut.indexOf("test song1") != -1);
  }

  public void testRelated2() {
    ...
    TEST_SONG1.showRelated();
    String strOut = baos.toString().trim();
    assertTrue(strOut.indexOf
      ("No related songs.") != -1);
  }

  public void testPlaylist() {
    ...
    Playlist p1 = new Playlist("my playlist");
    ...
    p1.play();
    String strOut = baos.toString().trim();
    assertTrue(strOut.indexOf
      ("playing album my playlist") != -1);
    assertEquals(0, user.getAccount().getBalance());
  }

  public void testPlaylistRemove() {
    ...
    p1.add(TEST_SONG1);
    p1.remove(TEST_SONG1);
    ...
    try {
      p1.play();
      fail();
    } catch (RuntimeException e) {}
  }

  public void testSession() {
    ...
    session.putValue("currentUser", user);
    session.removeValue("custom user");
    assertNull(session.getValue("custom user"));
  }
}
```

Figure 7. Test set adequate for the structural unit testing criteria, for the Music Online application.

To use the pointcut-based criteria we can run the JaBUTi/PC-AJ tool (presented in the next section). We can import the unit test set presented above and see the coverage for each criterion, checking how the `bill` advice is being covered at each join point (according to the $\mathcal{PCCFG}$ graph presented in Figure 6). The testing requirements derived for this example are presented in Table 1.

When we import the test set in the JaBUTi/PC-AJ tool, some of the `bill` advice



Overall Coverage Summary by Criterion

Testing Criterion	Coverage	Percentage
All-Nodes-ei	51 of 51	100%
All-Nodes-ed	0 of 0	0%
All-Edges-ei	22 of 22	100%
All-Edges-ed	0 of 0	0%
All-Uses-ei	34 of 34	100%
All-Uses-ed	0 of 0	0%
All-Nodes-c	4 of 4	100%
All-Edges-c	6 of 6	100%
All-Uses-c	1 of 1	100%

Figure 8. Coverage obtained after executing the 14 test cases of the unit testing test set.

Table 1. Set of requirements derived by the pointcut-based integration testing criteria for the `bill` advice and corresponding pointcut.

Criterion	Requirements
all-pc-nodes	R_n = { 1:0, 1:4, 1:7, 1:8, 1:11, 1:16, 1:24, 1:30, 1:33, 1:36, 1:46, 1:45, 1:50, 1:54, 2:0, 2:4, 2:7, 2:8, 2:11, 2:16, 2:24, 2:30, 2:33, 2:36, 2:46, 2:45, 2:50, 2:54, 3:0, 3:4, 3:7, 3:8, 3:11, 3:16, 3:24, 3:30, 3:33, 3:36, 3:46, 3:45, 3:50, 3:54, 4:0, 4:4, 4:7, 4:8, 4:11, 4:16, 4:24, 4:30, 4:33, 4:36, 4:46, 4:45, 4:50, 4:54 }
all-pc-edges	R_e = { (1:0,1:4), (1:4,1:7), (1:4,1:8), (1:8,1:11), (1:11,1:16), (1:16,1:24), (1:24,1:30), (1:30,1:33), (1:33,1:36), (1:36,1:45), (1:33,1:46), (1:46,1:50), (1:50,1:54), (2:0,2:4), (2:4,2:7), (2:4,2:8), (2:8,2:11), (2:11,2:16), (2:24,2:30), (2:30,2:33), (2:33,2:36), (2:36,2:45), (2:33,2:46), (2:46,2:50), (2:50,2:54), (3:0,3:4), (3:4,3:7), (3:4,3:8), (3:8,3:11), (3:11,3:16), (3:16,3:24), (3:24,3:30), (3:30,3:33), (3:33,3:36), (3:36,3:45), (3:33,3:46), (3:46,3:50), (3:50,3:54), (4:0,4:4), (4:4,4:7), (4:4,4:8), (4:8,4:11), (4:11,4:16), (4:16,4:24), (4:24,4:30), (4:30,4:33), (4:33,4:36), (4:36,4:45), (4:33,4:46), (4:46,4:50), (4:50,4:54) }

nodes and edges are not covered at some of the join points, that is, the test set is not adequate for the all-pc-nodes criterion – 70% of the advice nodes are covered (see Figure 11(c)) – neither for the all-pc-edges criterion. For instance, at the `showComposer` join point, only the part of the advice that returns without billing due to a bonus song is executed (path 0–4–7 of the $\mathcal{AODU}$ showed in Figure 5; more specifically, path 3.0–3.4–3.7 in the $\mathcal{PCCFG}$ showed in Figure 6). This happens because the test case created to test `showComposer` was called against a bonus song (test case `testShowComposer` at lines 90–92 of the test set code in Figure 7). At this point, the tester is forced to create additional test cases to adequate the initial test set to the pointcut-based criteria. Some of these test cases must execute the remaining parts of the `bill` advice at the `showComposer` join point. The tester is then led to discover a fault present in the program: the `showComposer` method should not be affected by the `bill` advice, because it refers to a non-chargeable operation (see the basic requirements for the Music Online application presented in Section 2.1). The fault is related to the pointcut descriptor, that should not affect all `Song` methods that start with `show`, because it includes the `showComposer` method. The error-revealing test case that executes some of the remaining parts of the `bill` advice at the `showComposer` context and fails is presented in Figure 9. When it executes, the assertion fails.

The activity should go on until 100% of the advice nodes and edges are covered at each join point, or until some coverage percentage set by a test plan is attained. In the end, five additional test cases are required to adequate the initial unit test set to the

```
public void testComposerToCoverCharge() {
  User user = new User("user1");
  Session session = Session.instance();
  session.putValue("currentUser", user);

  user.getAccount().credit(11);
  TEST_SONG1.showComposer();
  assertEquals(11, user.getAccount().getBalance());
}
```

Figure 9. Error-revealing test case created to adequate the initial test set to the pointcut-based testing criteria.

pointcut-based criteria. No additional faults are found.

Note that although the initial test set was adequate for the structural unit testing criteria referred to before, it was not able to discover the fault. This happens because the unit testing criteria do not force the tester to cover the advice statements at each join point, it only forces it to be covered by whatever means. On the other hand, the pointcut-based criteria do require a test case that necessarily uncovers the fault[3].

5. Implementation

The JaBUTi family of tools [Vincenzi et al. 2006, Lemos et al. 2007] is currently being extended to make it possible the use of the pointcut-based testing approach for AspectJ programs. We explain the extension, called JaBUTi/PC-AJ (for pointcut-based AspectJ), in four parts: (1) the identification of the pairs of units related to pieces of advice in the program, that is, the identification of join points for a specific advice-pointcut pair; (2) the generation of the $\mathcal{PCCFG}$ graph; (3) the implementation of the criteria; and (4) the implementation of the pointcut-based testing environment. The extension is being made on top of the pairwise integration testing version of JaBUTi – the PW-AJ version [Franchin et al. 2007]. In fact, the extension is already functional, missing only the displaying of the $\mathcal{PCCFG}$ graphs. In any case, the graphs of the corresponding pairs can be seen separately, such as in the PW-AJ version.

5.1. Identification of join points for specific pieces of advice

To implement the pointcut-based testing model and criteria defined earlier, we need to identify all interacting unit pairs for a particular advice, where the base unit contains a join point related to that advice. For instance, for the example presented in Section 2, to construct the $\mathcal{PCCFG}$ presented for the `bill` advice and related pointcut, we need to detect which methods (or possibly pieces of advice) are affected by the advice. We can use the AspectJ model depicted in Figure 10 to understand the idea more clearly. An aspect can contain several pointcuts and several pieces of advice and an advice is linked to a single pointcut[4]. A pointcut selects several join points which in turn are located at specific units.

[3]The tester could indeed discover the fault during the unit testing phase, or even just by looking at the $\mathcal{AODU}$ of the `showComposer` method – which shows that the advice is being executed where it should not –, but not necessarily. That is, there are possible test sets (including the one in Figure 7) that are adequate for the unit testing criteria, but do not uncover the fault.

[4]The pointcut itself can be a composition of other pointcuts, but we do not model this type of composition here. Consider the pointcut as the final composition used by a particular advice.

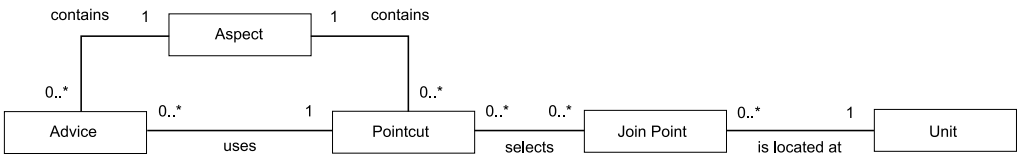


Figure 10. Simplified partial AspectJ model.

An `AdvicePointcut` class maintains a list of pieces of advice and related pointcuts, that is, it models the relation between *Advice* and *Pointcut*. For each instance of this class, we maintain a list of units that contain the related join points (that is, the relation between *Join Point* and *Unit*). To create the `AdvicePointcut` instances, since the JaBUTi family is based on bytecode, we analyze the bytecode itself. To discover pieces of advice, we use functions provided by the AspectJ compiler, that also work at bytecode level. The process of identifying join points is simplified because the possible interactions between pieces of advice and other units are already resolved at the bytecode level (*i.e.* the program is already woven). Therefore, we only need to check all bytecode level method calls that represent join points, and group them by advice.

5.2. Generation of the $\mathcal{PCCFG}$ graph

To construct the $\mathcal{PCCFG}$ graph, we need to gather all pairs of interacting units for each advice-pointcut pair. The JaBUTi/PW-AJ version already generates graphs for each pair of unit; therefore, we only need to group them by advice in a single graph, and construct the nodes that represent the entry and exit points from the affected execution regions. We also need to add edges from the $\mathcal{PCCFG}$ entry point node to the entry nodes of all of the base units, and from the exit nodes of these units to the $\mathcal{PCCFG}$ exit point node.

5.3. Implementation of the pointcut-based integration testing criteria

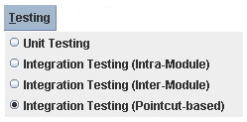
The implementation of the two control-flow pointcut-based criteria is done on top of the integration pairwise criteria implemented in JaBUTi/PW-AJ. The idea is the following: since the pairwise criteria already gather requirements for each pair of interacting units, for the pointcut-based criteria we only need to group the requirements of a set of pairs for each advice-pointcut pair. For instance, consider the all-pc-nodes criterion and the example presented on Section 2. The requirements for the all-pairwise-integrated-nodes [Franchin et al. 2007] criterion include the nodes of the `bill` advice in each place it affects the program, however, the requirements are derived by pair of units, separately. To gather the requirements for the all-pc-nodes criterion, we can collect the requirements of all the unit pairs that have the `bill` advice as the integrated unit. The same works for the all-pc-edges, but in this case using the all-pairwise-integrated-edges criterion as a basis.

5.4. Implementation of the pointcut-based testing environment

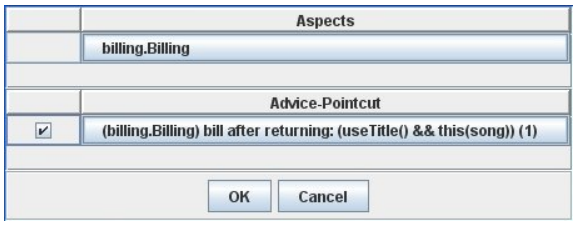
To support the pointcut-based testing approach, we need an additional environment to the JaBUTi/PW-AJ tool. This environment uses the same modules selected to be tested and instrumented in the unit testing environment – the initial environment of JaBUTi – and, from these modules, we can identify the pieces of advice and all their interactions in the program.

The pointcut-based testing environment supports its specific testing data so that all that happens in this environment does not affect the others. An example is the execution of test cases. While executing a particular test case in the unit testing environment, for instance, we do not want such execution to affect the integration testing information. The tester can also save the testing project and the separation among the environments' information will be kept for later usage of the same project.

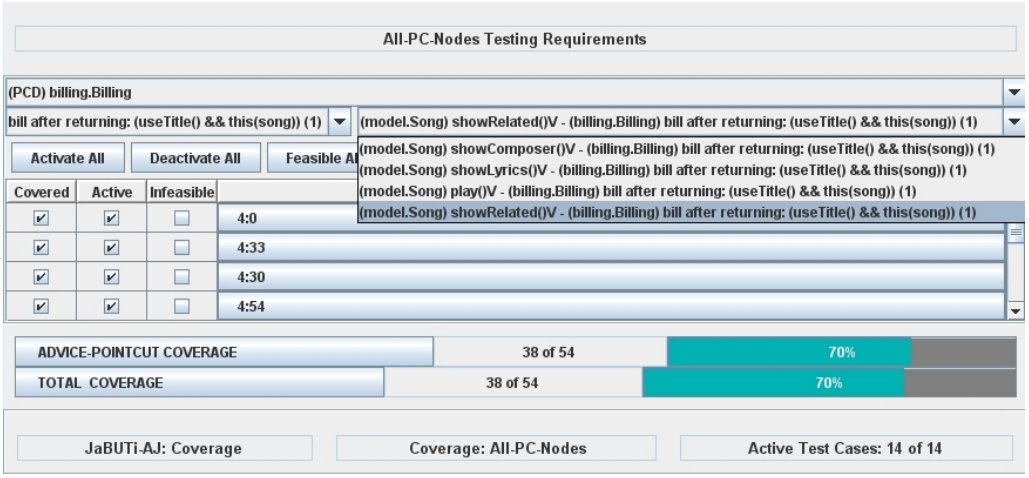
The pointcut-based testing environment supports the following activities: checking of the requirements derived for each criterion, importing of JUnit test cases, checking of the coverage obtained by the imported test cases, and visualizing the graphs for each advice-pointcut pair. Figure 11 presents some screenshots of the pointcut-based testing environment while testing the example presented in Section 2.



(a) Pointcut-based testing selection.



(b) Selecting the advice-pointcut to be tested.



(c) Requirements and coverage of the `bill` advice and corresponding pointcut for the All-PC-nodes criterion.

Figure 11. Screenshots of the JaBUTI/PC-AJ tool.

6. Related Work

To the best of our knowledge few testing criteria were defined for integration testing of Object-Oriented (OO) programs, and even fewer for the integration testing of AO programs. Zhao has developed a data-flow testing approach for AO programs [Zhao 2003] based on the OO approach proposed by Harrold and Rothermel [Harrold and Rothermel 1994]. He also addresses the testing of interfaces between class units and aspect units, but does not limit the depth of interactions. Also, Zhao does not focus on the advice interactions: the presented model treats methods and pieces of advice indiscriminately, which makes it harder to separately reason about the advice coverage. Moreover, until now, no implementation of the approach has been presented.

Franchin et al.[Franchin 2007] explored the pairwise integration testing of OO and AO Java programs (see Section 2.2). A limitation of this approach is that pairs of methods and advice are treated indiscriminately. That is, no particular attention is given to the integration of crosscutting concerns implemented as advice at join points: these pairs are treated as method pairs. Moreover, to create test sets that are adequate for each of the pairwise criteria, the tester must cover all types of pairs (*e.g*, all methods being called by other methods), which causes the activity to be more expensive. Also, a tester concerned with the global coverage of an advice that affects several join point must check each related pair, because there is no specific coverage measure for pieces of advice.

7. Conclusion

In this paper we have presented a pointcut-based integration testing approach for AO programs. A model to represent the execution regions of a program affected by pieces of advice was proposed and two control-flow based criterion were defined. To present evidence of the efficiency of our approach, we presented an example where a test set adequate for unit testing criteria can be constructed but still not reveal a pointcut-related fault present in the program. We showed that a test set adequate for the pointcut-based criteria necessarily uncovers the fault.

We believe the pointcut-based criteria can also help in other testing activities for AO programs. For instance, while regression testing programs that are added with aspects, testers should focus on the points affected by these additions. Since the $\mathcal{PCCFG}$ models exactly these regions of the program, the defined criteria can be used to enhance the confidence that these regions still work as originally intended.

A limitation of our approach we need to explore is the scalability. Pieces of advice that affect a large amount of execution regions of the program generate large $\mathcal{PCCFG}$s and, consequently, large sets of requirements. We are currently working on solutions for this problem. For instance, the tool could generate warnings for the user when an advice affects a large number of join points (we also need to quantify what would be a 'large' set of join points). In this case, the user could have the possibility to select subsets of join points to represent the real set, and analyze the coverage for this particular subset. Future work also includes an evaluation of the application of the criteria on larger applications, to make a deeper analysis of their efficiency and cost of application.

References

Bertolino, A. (2007). Software testing research: Achievements, challenges, dreams. In *FOSE '07: 2007 Future of Software Engineering*, pages 85–103, Washington, DC, USA. IEEE Computer Society.

Elrad, T., Kiczales, G., Akşit, M., Lieberher, K., and Ossher, H. (2001). Discussing aspects of AOP. *Communications of the ACM*, 44(10):33–38.

Franchin, I. G. (2007). Teste estrutural de integração par-a-par de programas orientados a objetos e a aspectos: Critérios e automatização. Master's thesis, ICMC-USP, São Carlos, SP.

Franchin, I. G., Lemos, O. A. L., and Masiero, P. C. (2007). Pairwise structural testing of object and aspect-oriented Java programs. In *Proceedings of the 21st Brazilian*

Symposium on Software Engineering, pages 377–393, Porto Alegre, RS, Brasil. SBC Press.

Harrold, M. J. and Rothermel, G. (1994). Performing data flow testing on classes. In *SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 154–163, New York, NY, USA. ACM Press.

Kiczales, G., Irwin, J., Lamping, J., Loingtier, J.-M., Lopes, C., Maeda, C., and Menhdhekar, A. (1997). Aspect-oriented programming. In Akşit, M. and Matsuoka, S., editors, *Proceedings of the ECOOP*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York. Springer-Verlag.

Kiczales, G. and Mezini, M. (2005). Aspect-oriented programming and modular reasoning. In *Proceedings of the* 27^{th} *International Conference on Software Engineering (ICSE'2005)*, pages 49–58. ACM Press.

Lemos, O. A. L., Ferrari, F. C., Masiero, P. C., and Lopes, C. V. (2006). Testing aspect-oriented programming pointcut descriptors. In *WTAOP '06: Proceedings of the 2nd workshop on Testing aspect-oriented programs*, pages 33–38, New York, NY, USA. ACM Press.

Lemos, O. A. L., Franchin, I. G., and Masiero, P. C. (2008). Integration testing of object-oriented and aspect-oriented programs: a structural pairwise approach for Java. (submitted for publication).

Lemos, O. A. L., Vincenzi, A., Maldonado, J. C., and Masiero, P. C. (2007). Control and data-flow structural testing criteria for aspect-oriented programs. *Journal of Systems and Software*, 80(6):862–882.

Mortensen, M. and Alexander, R. T. (2005). An approach for adequate testing of AspectJ programs. In *Proceedings of the* 1^{st} *Workshop on Testing Aspect Oriented Programs – in conjunction with AOSD'2005*, Chicago/IL, USA.

Rapps, S. and Weyuker, E. J. (1985). Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.*, 11(4):367–375.

Vincenzi, A. M. R., Delamaro, M. E., Maldonado, J. C., and Wong, W. E. (2006). Establishing structural testing criteria for java bytecode. *Softw. Pract. Exper.*, 36(14):1513–1541.

Vincenzi, A. M. R., Maldonado, J. C., Wong, W. E., and Delamaro, M. E. (2005). Coverage testing of java programs and components. *Science of Computer Programming*, 56(1–2):211–230.

Zhao, J. (2003). Data-flow-based unit testing of aspect-oriented programs. In *Proceedings of the* 27^{th} *Annual IEEE International Computer Software and Applications Conference (COMPSAC'2003)*, pages 188–197, Dallas/Texas - USA. IEEE Computer Society.