Control and Data Flow Structural Testing Criteria for Aspect-Oriented Programs

Otávio Augusto Lazzarini Lemos[a], Auri Marcelo Rizzo Vincenzi[b], José Carlos Maldonado[a], Paulo Cesar Masiero[a]

[a] *Universidade de São Paulo*
Instituto de Ciências Matemáticas e de Computação
Av. do Trabalhador São-Carlense, 400 – São Carlos, SP

[b] *Instituto de Informática – Universidade Federal de Goiás*
Bloco IMF I, sala 239 – Campus II – Samambaia – Caixa Postal 131 – CEP 74001-970 – Goiânia, GO

Abstract

Although it is claimed that, among other features, aspect-oriented programming (AOP) increases understandability and eases the maintenance burden, this technology cannot provide correctness by itself, and thus it also requires the use of systematic verification, validation and testing (VV&T) approaches. With the purpose of producing high quality software, many approaches to apply structural testing criteria for the unit testing of procedural and object-oriented (OO) programs have been proposed. Nevertheless, until now, few works have addressed the application of such criteria to test aspect-oriented programs. In this paper we define a family of control flow and data flow based testing criteria for aspect-oriented programs inspired by the implementation strategy adopted by AspectJ – an aspect-oriented extension of the Java language – and extending a previous work proposed for Java programs. We propose the derivation of a control and data flow model for aspect-oriented programs based upon the static analysis of the object code (the Java bytecode) resulted from the compilation/weaving process. Using this model, called aspect-oriented def-use graph ($\mathcal{AODU}$), traditional and also aspect-oriented testing criteria are defined (called Control and Data Flow Structural Testing Criteria for Aspect-Oriented Programs - CDSTC-AOP). The main idea is that composition of aspect-oriented programs leads to new crosscutting interfaces in several modules of the system, which must be considered for coverage during structural testing. The implementation of a prototype tool – the JaBUTi/AJ tool – to support the criteria and the model proposed is presented along with an example. Also, theoretical and practical questions regarding the CDSTC-AOP criteria are discussed.

Key words: Software Testing, Aspect-Oriented Programming, Structural Testing, Testing Criteria, Testing Aspect-Oriented Programs

1 Introduction

Most programming languages do not support the clear separation of some types of requirements – usually non-functional – that tend to be spread throughout several modules of implementation (for instance, the implementation of logging). Aspect-oriented programming (AOP) supports the modularization of such concerns by mechanisms that make possible the addition of behavior – *advisement* – to selected elements of the programming language semantics (the *join points*), thus isolating implementation that would otherwise be spread throughout the base code. This mechanism supports the development of programs whose structure more closely reflect their designs.

Until recently, research in the field has focused on the establishment of AOP underlying concepts and aspect-oriented (AO) language implementations and applications. Recently, however, researchers are turning to other issues related to aspect oriented software development (AOSD), such as AO software design and management of aspects during requirements engineering [5,25]. Software testing is also a topic of interest in this new phase.

It has been claimed that software developed using AOP tend to be more understandable and easier to maintain, based on a more effective use of the separation of concerns principle. However, such technique cannot provide correctness by itself, since it does not prevent developers from introducing errors in the system, during development [40]. In fact, it has been argued that AOP can introduce even more sources of fault compared to traditional software, mainly because of its powerful mechanisms [2,4,21,22]. Hence verification, validation and testing (VV&T) techniques are still important in the AOSD process and might also require additional adaptations to be effectively used in this context.

One of the existing testing approaches is the structural testing technique (also known as *white-box* testing), which derives test cases from the logical structure of a program. The main idea of the technique is that one cannot trust a piece of software if there are still certain elements in its structure that were never executed during testing. Some examples of structural testing criteria require that the test cases cover: all statements (or all-nodes), all branches (or all-edges), and all variable definition-use pairs of a program [43,24].

In the effort to produce high quality applications, many approaches to apply structural testing criteria for unit and integration testing of procedural and OO programs have been proposed [12,24,32,33]. Nevertheless, until now, few works have addressed the application of structural testing criteria to AO programs. The structural testing technique is usually applied for unit testing although research on using it in integration testing can also be found [11,12,19].

Zhao [40,41] was the first researcher that presented a strictly structural testing approach for AO programs. In his work a data flow unit testing approach to test AO programs based on the AspectJ language inspired by the Harrold and Rothermel approach for class testing [12] is proposed. Although called "unit" testing, the approach also considered the interactions among classes and aspects (because he considered *clustered* aspects and classes as the units to be tested).

Alexander et al. [4] and Mortensen & Alexander [22] also propose coverage criteria for AO programs. Two testing criteria are discussed: *Insertion* coverage which means testing each aspect code fragment at each point it is woven into the program (this criterion is similar to the all-crosscutting-nodes criterion that will be defined in this paper); and *Context* coverage which extends insertion coverage to test an aspect code fragment in each place it is used, by requiring, for instance, the testing of each method that a piece of advice defines behavior. Mortensen and Alexander [22] also discuss the application of mutation testing to test the quantification mechanism of AO languages.

Xu et al. [36,38] propose the application of state based testing for AO programs. The main idea is to extend the FREE model proposed by Binder [7] in an *Aspectual State Model* (ASM) for AO programs. In the second part of the work it is also proposed the use of structural testing along with the state based approach, by replacing the transitions and advice interactions with the corresponding control flow graphs, constructing an *Aspect Flow Graph* (AFG). However, coverage analysis is briefly discussed. Recently, the same researchers proposed a model-based approach to generate test cases for AO programs [37].

Some papers address the testing of aspects in isolation, from the perspective of trying to execute and test pieces of advice and quantification mechanisms without the need to weave the aspects with the rest of the system. Lopes et al. [20] propose the extension of the JUnit framework to the unit testing of aspect behavior implemented using the JAML language. Yamazaki et al. [39] propose a framework to test aspects in isolation, addressing advice block testing and also the testing of the quantification mechanism. These approaches could be used together with the one that will be presented in this paper so that the pieces of advice can be executed and analyzed in isolation, without the need of any base code.

Although few research on applying testing techniques to AOSD can be found, some works have been proposed towards applying statical analysis techniques in this context. Störzer et al. [27,28], for example, proposes the application of program analysis to AO programs in order to detect some types of flaws particular to such programs. Denaro and Monga [8] proposed the derivation of models for AO programs, suitable for verifying system properties (a similar work is presented by Ubayashi and Tamai [30]).

In this paper we propose the derivation of a model to represent the control and data flow of aspect-oriented programs implemented in AspectJ to support an adequate structural testing approach for such programs. Based on this model traditional testing criteria are defined along with a new family of aspect-oriented testing criteria (we call both traditional and aspect-oriented criteria *Control and Data Flow Structural Testing Criteira for Aspect-Oriented Programs* – CDSTC-AOP). The main idea is that inspired by recent versions of AspectJ and working with the woven artifacts, we can have an approach that implies in a more direct application of structural testing criteria and also support the peculiarities present in the structure of AO programs. For this purpose we extend the work of Vincenzi et al. [32–34], that have proposed control and data flow models for OO programs based on the Java bytecode.

The remainder of this paper is organized as follows. Section 2 briefly introduces the aspect-oriented programming main concepts, the AspectJ language and its implementation strategy. Section 3 presents an underlying control and data flow model for AspectJ programs based on its implementation. Section 4 presents our approach for structural unit testing of AO programs, defining the CDSTC-AOP for Java bytecode aiming at testing aspect-oriented programs. Section 5 presents an inclusion relation analysis among the CDSTC-AOP criteria based on the subsumes relation [10,42,43]. Section 6 gives an example of use of the approach, presenting the JaBUTi/AJ prototype tool and Section 7 presents a preliminary study on the efficacy and application cost of the proposed criteria. Finally, Section 8 presents our conclusions and further directions.

2 Aspect Oriented Programming and the AspectJ language

The AOP main idea is that while OO programming, procedural and other programming techniques by themselves help separating out the different concerns implemented in a software system, there are still some requirements (usually non-functional) that cannot be clearly mapped to isolated units of implementation. Examples of those concerns are mechanisms to persist objects in relational data bases, access control, quality of services that require fine tuning of system properties, synchronization policies and logging. These are often called *crosscutting* concerns, because they tend to cut across multiple elements of the system instead of being localized within specific structural pieces [9].

AOP then support the construction of separate units – the aspects – that have the ability to cut across the system units, defining behavior that would otherwise be spread throughout the base code. A generic AOP language should then define: a model to describe hooks in the base code where additional behavior may be defined (these hooks are called *join points* which are well-defined points in the execution of a program [29]); a mechanism of identification of these join points; units that encapsulate both join point specifications and behavior enhancements; and a process to combine both base code and aspects (which is called the *weaving* process) [9].

2.1 The AspectJ Language

AspectJ is an extension of the Java language to support AOP. The basic new constructs are the `aspect` itself; `before`, `after` and `around` advice, that are used to define crosscutting behavior at join points; and `pointcut`s which are used to define sets of join points in the program.

`Aspect`s are units that combine: join point specifications (points in the system where additional behavior may be defined), pieces of advice, which are the actual desired behavior to be defined and methods, fields and inner classes. Also, aspects can declare members (fields and methods) to be owned by other types, what is called inter-type declarations. Recent versions of AspectJ also support declarations of warnings and errors that arise when join points are identified or reached [29]. Before, after and around advice are method-like constructs that can be executed before, after and in place of the join points, respectively. These constructs can also pick context information from the join point that has caused them to execute. Figure 1 lists part of the source code (one class and two aspects) of an aspect-oriented program that will be used along this paper. The application simulates a telephony system and is an extended version of the telecom application that comes with the AspectJ distribution [29]. The application has been altered to support a different type of charging for mobile calls.

The classes of the system are:

- `Customer`, which has name and area code fields and models customers;
- `Connection` (which is abstract) and two concrete classes `Local` and `LongDistance`, that model local and long distance connections;
- `Call`, which models telephone calls (illustrated in Figure 1);
- `Timer`, which models timers.

The aspects of the system are:

- `Timing`, which implements the timing concern and measures the connection duration for the customers, initializing and stopping a timer associated with each connection (Figure 1);
- `Billing`, which implements the billing concern and declares a payer to each connection and also makes sure that local, long distance and mobile calls

```
public class Call {
    private Customer caller, receiver;
    private Vector connections = new Vector();

    public Call(Customer caller, Customer receiver,
                boolean iM)
    {
01     this.caller = caller;
02     this.receiver = receiver;
03     Connection c;
04     if (receiver.localTo(caller)) {
05     c = new Local(caller, receiver, iM);
06     } else {
07     c = new LongDistance(caller,
08     receiver, iM);
09    }
10    connections.addElement(c);
    }

    public void pickup() {
        Connection connection =
        (Connection)connections.lastElement();
        connection.complete();
    }

    public boolean isConnected(){
        return
        ((Connection)
          connections.lastElement()).getState()
        == Connection.COMPLETE;
    }

    public void hangup(Customer c) {
        for(Enumeration e =
            connections.elements();
            e.hasMoreElements();) {
        ((Connection)e.nextElement()).drop();
        }
    }

    public boolean includes(Customer c){
        boolean result = false;
        for(Enumeration e =
            connections.elements();
            e.hasMoreElements();) {
            result = result ||
            ((Connection)
              e.nextElement()).connects(c);
        }
        return result;
    }

    public void merge(Call other){
        for(Enumeration e =
            other.connections.elements();
            e.hasMoreElements();){
            Connection conn =
              (Connection)e.nextElement();
            other.connections.removeElement(conn);
            connections.addElement(conn);
        }
    }
}

public aspect Timing {
    public long Customer.totalConnectTime = 0;

    public long
      getTotalConnectTime(Customer cust) {
        return cust.totalConnectTime;
    }

    private Timer Connection.timer =
      new Timer();

    public Timer getTimer(Connection conn) {
      return conn.timer;
    }
    after (Connection c) returning () :
      target(c) && call(void
      Connection.complete()) {
        getTimer(c).start();
    }

    pointcut endTiming(Connection c): target(c) &&
        call(void Connection.drop());

    after(Connection c) returning () :
      endTiming(c) {
        getTimer(c).stop();
        c.getCaller().totalConnectTime +=
          getTimer(c).getTime();
        c.getReceiver().totalConnectTime +=
          getTimer(c).getTime();
    }
}

public aspect Billing {
    declare precedence: Billing, Timing;

    public static final long LOCAL_RATE = 3;
    public static final long
      LONG_DISTANCE_RATE = 10;
    public static final long
      MOBILE_LD_RECEIVER_RATE = 5;

    public Customer Connection.payer;
    public Customer getPayer(Connection conn)
    { return conn.payer; }

    after(Customer cust) returning
      (Connection conn): args(cust, ..)
      && call(Connection+.new(..))
    { conn.payer = cust; }

    public abstract long Connection.callRate();

    public long LongDistance.callRate() {
        return LONG_DISTANCE_RATE;
    }

    public long Local.callRate() {
        return LOCAL_RATE;
    }

    after(Connection conn) returning () :
        Timing.endTiming(conn) {
        long time =
        Timing.aspectOf().getTimer(conn).getTime();
        long rate = conn.callRate();
        long cost = rate * time;
        if (conn.isMobile()) {
          if (conn instanceof LongDistance) {
            long receiverCost =
              MOBILE_LD_RECEIVER_RATE * time;
            conn.getReceiver().addCharge
            (receiverCost);
          }
        }
        getPayer(conn).addCharge(cost);
    }

    public long Customer.totalCharge = 0;
    public long getTotalCharge(Customer cust) {
      return cust.totalCharge;
    }

    public void Customer.addCharge(long charge){
        totalCharge += charge;
    }
}
```

Fig. 1. Extended version of an AO program example [29] written in AspectJ.

are charged accordingly (Figure 1);

- `TimerLog`, which implements a log that prints the times when the timer starts and stops.

In any AOP language implementation, aspect and non-aspect code must run in a properly coordinated fashion. In order to do so an important issue is to ensure that pieces of advice run at the appropriate join points as specified by the program. Previous versions of AspectJ used the strategy of inlining the advice code directly into the join points, which resulted in `.class` files that were a mixture of aspect and non-aspect code [13]. In fact, older versions of AspectJ did the weaving process based on source code: first files were pre-processed into standard Java and then these new files were compiled with a standard compiler. The recent AspectJ advice weaver is based on bytecode, so this process is made by bytecode transformation rather than on source code files.

The AspectJ advice weaver statically transforms the program so that at run-time it behaves according to the language semantics. The compiler accepts both AspectJ bytecode and source code and produces pure Java bytecode as a result. The main idea is to compile the aspect and advice declarations into standard Java classes and methods (at bytecode level). Parameters of the pieces of advice are now parameters of these new methods (with some special treatment when reflexive information is needed) [13]. In order to coordinate aspects and non-aspects the system code is instrumented and calls to the "advice methods" are inserted considering that certain regions of the bytecode represent possible join points (these are called join point *static shadows* [13]). Furthermore if the join point cannot be completely determined at compile time, these calls are guarded by dynamic tests to make sure that the pieces of advice run only at appropriate time (these tests are called *residues*)[1] [13].

In older versions of AspectJ, pieces of advice were injected directly into the methods of classes, resulting in blocks that mixed aspect with non-aspect code, making it difficult to distinguish one from another. In recent versions, however, methods are only added with method calls – possibly guarded by dynamic tests – to advice methods before, after or in place of the corresponding join points, depending on the advice type. Thus, woven classes generated with current versions are cleaner than before and we can also identify the join points, where aspects are defining behavior and also which pieces of advice are executing at those join point (identified by the advice method names). Also, as aspects are compiled into separate aspect classes, they may be instrumented and analyzed separately supporting separate coverage analyses of pieces of advice and methods of aspects.

With this implementation strategy one can thus identify the places where

[1] The AspectJ compiler may also inline some advice in special cases but the `-XnoInline` option can be used to prevent it from using that strategy [13].

pieces of advice are adding behavior through a static analysis of the bytecode resulted from the compilation/weaving process. That is, every call that is made to an advice method in the bytecode represents an execution of the corresponding advice of some aspect in the affected point.

Figure 2 shows part of the bytecode and the aspect-oriented definition-use ($\mathcal{AODU}$) graph (see Section 3) generated by the JaBUTi/AJ tool to represent the unit control and data flow of the `Call` class constructor presented in Figure 1, based on the bytecode (data flow information is implicit and can be viewed by pointing the mouse on each node of the graph). The labels of regular nodes are the numbers of the first bytecode instructions of the corresponding blocks. Dashed nodes represent advice invocations (*crosscutting nodes* [17,18]) and have additional information about the type of advice that affects that point, and the name of the aspect it belongs to (for instance $\ll afterReturning - telecom.Billing \gg$ corresponds to an after returning advice of the `Billing` aspect). Bold nodes represent exit nodes, double-circled nodes represent method calls and dashed edges represent abnormal control flow when exceptions are thrown (not presented in the example).

To understand how the $\mathcal{AODU}$ is constructed based on bytecode, looking at Figures 1 and 2, note that the block of bytecode instructions 0–1 does not have correspondence with the source code (it represents a call to method `Object` which is part of any constructor in Java bytecode); the block of bytecode instructions 4–30 corresponds to lines 1–4 of the constructor's source code; the block of bytecode instructions 33–69 corresponds to line 5 of the constructor's source code and, since an advice runs at calls to the `Local` constructor, a crosscutting node is used to represent that block. The block of bytecode instructions 78–114 corresponds to lines 7–8 of the constructor's source code and, since an advice runs at calls to the `LongDistance` constructor as well, another crosscutting node is used. The blocks of bytecode that start in instructions 72 and 117 do not have correspondence with the source code since they just have `nop` and `goto` statements; and block of bytecode instructions 120–129 corresponds to line 10 of the constructor's source code.

Note that the labels of the nodes are the numbers of the first bytecode instructions of the corresponding blocks and that the logic of the $\mathcal{AODU}$ is straightforward when we inspect both source or bytecode. For instance, it is clear why there is two edges coming out from node 4 since there is an *if-then-else* conditional in the consructor's code. Besides, the mapping from source code to bytecode, and from bytecode to the $\mathcal{AODU}$ is explicit in the JaBUTi/AJ tool, which will be presented later. A formal definition of the graph will also be presented later.

```
  0 aload_0
  1 invokespecial #15 <Method Object()>
  4 aload_0
  ...
 27 invokevirtual #30 <Method boolean
    localTo(telecom.Customer)>
 30 ifeq 78
 33 aload_1
 ...
 52 invokespecial #34 <Method
    Local(telecom.Customer,
    telecom.Customer, boolean)>
 ...
 69 invokevirtual #110 <Method void
    ajc$afterReturning$telecom_Billing$
    1$8a338795(
    telecom.Customer, telecom.Customer,
    boolean, telecom.Connection)>
 72 nop
 73 astore 4
 75 goto 120
 78 aload_1
 ...
 97 invokespecial #37 <Method
    LongDistance(telecom.Customer,
    telecom.Customer, boolean)>
 ...
114 invokevirtual #110 <Method void
    ajc$afterReturning$telecom_Billing$
    1$8a338795(
    telecom.Customer, telecom.Customer,
    boolean, telecom.Connection)>
117 nop
118 astore 4
120 aload_0
121 getfield #20 <Field java.util.Vector
    connections>
124 aload 4
126 invokevirtual #41 <Method void
    addElement(java.lang.Object)>
129 return
```

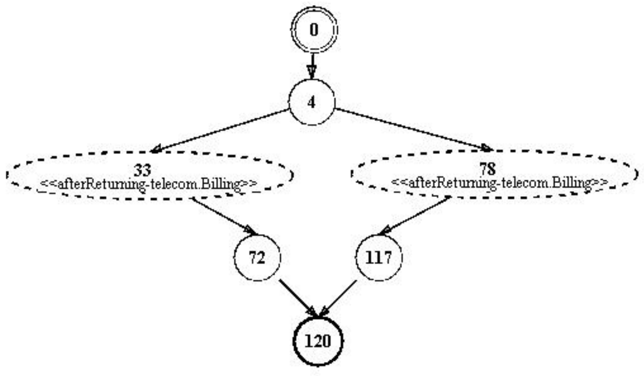

0
4
33
<<afterReturning-telecom.Billing>>
78
<<afterReturning-telecom.Billing>>
72
117
120

Fig. 2. Bytecode and $\mathcal{AODU}$ of the constructor of the `Call` class.

3 Underlying Control and Data Flow Models for AspectJ Programs

The most known program representation for establishing control flow testing criteria is the control flow graph (CFG). With respect to the data flow testing criteria, it is usually used the definition-use (or def-use) data flow graph, which is an extension of the CFG with additional information about the definition and use of variables in each node and edge of the CFG [24]. The def-use graph can also be used to derive control flow testing requirements, since it is an extension of the CFG.

Vincenzi et al. [32,33] have defined control and data flow models based on the Java bytecode for the unit testing of OO programs implemented with Java. With respect to AO programs written in AspectJ, such approach remains interesting because, as pointed before, the compiler produces pure Java bytecode that can be analyzed directly. However, such models must be extended to adequately represent the peculiarities present in the structure of AO programs.

The main intra-unit difference present in the structure of an AO program is the change of control flow from units to pieces of advice that occur when aspects define behavior on join points in those units. This phenomena happens because composition of aspect-oriented programs leads to new crosscutting interfaces in several methods of the system [14]. In AspectJ, this mechanism is implemented with insertions of method calls before, after or in place of the join points, as explained in the previous section. Thus, as these method calls can be detected by a static analysis of the bytecode, an adequate graph to represent the control flow of the units of such programs can be constructed based on such object code. The blocks where advice implicit invocations are identified are represented using special types of nodes which are called *crosscutting nodes.* These nodes are *crosscutting* because they can cut across several units, since a single advice can be applied to several join points located in those units.

3.1 The $\mathcal{AODU}$ graph

We propose the $\mathcal{AODU}$ data flow graph as the model from which control and data flow testing requirements are derived. This type of graph must be constructed for each method-like construct of AspectJ which are: regular methods and constructors, pieces of advice and inter-type declared methods and constructors. We consider these constructs as the smallest units of an AO program.

Before constructing the $\mathcal{AODU}$ graph, following Vincenzi et al. [32,33], we construct the data flow instruction graph ($\mathcal{IG}$) of each unit. Informally the $\mathcal{IG}$ is a graph whose nodes contain a single bytecode instruction and whose edges connect instructions that can be executed sequentially [2]. Furthermore, the $\mathcal{IG}$ is annotated with data flow information (definition and use of variables) for each bytecode instruction.

The idea of the $\mathcal{IG}$ graph is to abstract the control and data flow involved in each bytecode instruction individually. Some bytecode properties must be handled carefully during the control flow analysis. In particular we have to handle the intra-module subroutine calls that the Java Virtual Machine (JVM) implements (`jsr` and `jsr_w`) and the exception handling mechanism of Java. In the latter case, the problem is that the exception handlers (`catch` blocks) of Java are not executed from regular control flow, but only when exceptions are thrown. The solution adopted to reflect the real control flow involved in this case was to use two different kinds of edges: *regular edges* that represent the normal control flow, that is, when no exceptions are thrown; and *exception edges* which represent the control flow when exceptions occur (a similar approach is presented by Sinha and Harrold [26], however considering only explicitly generated exceptions) [31].

[2] Note that the residues (2.2) inserted by the AspectJ weaver found in the bytecode are also treated. Furthermore we will study whether these kinds of instructions should be highlighted in our model for showing, for instance, that the execution of a certain advice relies on dynamic information.

Formally, the $\mathcal{IG}$ graph of a unit u is defined as a directed graph $\mathcal{IG}(u) = (NI, EI, si, TI, CI)$, such that:

- NI represents the non-empty set of nodes of an $\mathcal{IG}$ graph: $NI = \{n_i | n_i$ corresponds to a bytecode instruction i, for all reachable bytecode i of u[3] $\}$.
- $EI = EI_r \cup EI_e$ is the complete set of edges of an $\mathcal{IG}$:
  - EI_r and EI_e correspond to two disjunct subsets of regular and exception edges, respectively;

    EI_r is the set of regular edges defined as $EI_r = \{(n_i, n_j)|$ the instruction in n_j can be executed immediately after the instruction in n_i and $(n_i, n_j) \notin EI_e\}$.

    EI_e is the set of exception edges defined as $EI_e = \{(n_i, n_j)|$ the instruction of n_i is in the scope of the exception handler[4] that begins in the instruction corresponding to the node n_j $\}$;
- $si \in NI$ is the entry node, which corresponds to the node that contains the first instruction of u. Consider x a node of a directed graph, $IN(x)$ corresponds to the number of incoming edges of x. Thus $IN(si) = 0$.
- $TI \subseteq NI$ is the set of exit nodes. Consider x a node of a directed graph, $OUT(x)$ corresponds to the number of outgoing edges of x. Thus $TI = \{n_i \in NI | OUT(n_i) = 0\}$.
- $CI \subseteq NI$ is the set of crosscutting nodes, *i.e.*, nodes that represent execution of pieces of advice from particular aspects. In fact, these nodes correspond to the advice methods invocation instructions (Section 2.2).

3.1.1 Data Flow Model

To compute data flow information, a data flow model must be defined. Such model will indicate which instructions correspond to definitions of variables and which correspond to uses of variables and how reference variables and arrays must be handled. In our case, an important step in the model definition is the classification of the bytecode instructions, that will relate each one with its implications on the data flow of the unit. The classification used for our model was taken from the work of Vincenzi et al. [32,33], which was defined to test OO programs. It has twelve classes of instructions, based on the data flow implications of each one. With such classification, data flow information about the definitions and uses of variables can be derived from the bytecode and be added to each node or edge.

[3] If the compiler ever generates unreachable bytecode instructions, such instructions are ignored in the construction of the $\mathcal{IG}$.

[4] An exception handler j is responsible for handling exceptions generated by the bytecode instructions located in an offset interval $[o_{j_m}..o_{j_n}]$ (a "`try`" block). A bytecode instruction i located in the offset o_i is in the scope of the j exception handler iff $o_i \in [o_{j_m}..o_{j_n}]$.

For the Java bytecode data flow model the following assumptions are also made to identify the types of data accesses treated:

(1) Array variables are considered as single memory locations and the definition/use of any of the elements of the array variable $a[]$ is considered as the definition/use of $a[]$. For example, in the command "$a[i] = a[j] + 1$" there is a definition and a use of array variable $a[]$;
(2) If an array variable $a[][]$ is declared, accesses to its elements are handled as definitions or uses of $a[]$, depending on the kind of access. Thus, in commands like "$a[0][0] = 10$" or "$a[0] = new\, int[10]$" there are definitions of $a[][]$ and $a[]$, respectively. In commands like "$a[0][0] = a[2][3]$" there is a use and a definition of $a[][]$.
(3) Every time an instance field is used/defined there is a use of the reference variable that allows the access to the field and a use/definition of the field itself. For example, considering two reference variables ref_1 and ref_2 to objects of class C that contains two instance integer (int) variables x and y, in the command "$ref\_1.x = ref\_2.y$" there are uses of reference variables ref_1 and ref_2, a use of the instance field $ref\_2.y$ and a definition of the instance field $ref\_1.x$. Moreover, every time an instance field is used and not defined in a unit, it is considered as defined in the first node of the corresponding $\mathcal{IG}$ graph;
(4) Class fields (static fields) are considered as global variables even if they are accessed via an object reference variable. The same assumption made for instance fields that are not defined within a unit is also made here;
(5) Method invocations like $ref\_1.foo(e\_1, e\_2, ..., e\_n)$ are considered a use of reference variable ref_1. Rules for definition/use identification on expressions e_1,$e\_2, ..., e\_n$ are the same described on assumptions 1 to 4. For instance units it is also considered a use of the current object **this** in the first node of the $\mathcal{IG}$. The same is considered for local variables that correspond to formal parameters of the executing unit. For class methods, only local variables corresponding to formal parameters are considered to be defined on the entry node of the $\mathcal{IG}$, from the fact that no instance variables are required to invoke a class method.

The $\mathcal{IG}$ graph offers a practical way of traversing the set of instructions in a given unit u, identifying the use and definition of variables. However, the number of nodes and edges involved in this type of graph may be too large to handle. Thus we construct the $\mathcal{AODU}$ based on the $\mathcal{IG}$ using the concept of *instruction block*, *i.e.*, instructions that are executed sequentially in a normal[5] control flow. When the first instruction of the block is executed, the following instructions are also executed. The $\mathcal{AODU}$ graph is then the base model to derive control and data flow testing requirements for the unit testing of As-

[5] *Normal* means not considering any possible interruptions that may break the execution of a program.

pectJ programs. An $\mathcal{AODU}$ graph of a given unit u is defined as a directed graph $\mathcal{AODU}(u) = (N, E, s, T, C)$, such that each node $n \in N$ represents an instruction block:

- N represents the set of nodes of a graph $\mathcal{AODU}$: $N = \{n|n$ corresponds to blocks of bytecode instructions of $u\}$, *i.e.*, N is the non-empty set of nodes, representing the bytecode instructions of u. I_n is the ordered n-tuple of instructions grouped in node n;
- $E = E_r \cup E_e$ is the complete set of edges of the $\mathcal{AODU}$ graph. Consider the $\mathcal{IG}(u) = (NI, EI, si, TI, CI)$:
  - · E_r is the set of regular edges defined as $E_r = \{(n_i, n_j)|$ exists a regular edge that goes from the last element of I_{n_i} to the first element of I_{n_j} in the $\mathcal{IG}(u)\}$;
  - · E_e is the set of exception edges defined as $E_e = \{(n_i, n_j)|$ exists an exception edge that goes from the last element of I_{n_i} to the first element of I_{n_j} in the $\mathcal{IG}(u)\}$;
  - · $E_c \subseteq E_r$ is the set of crosscutting edges defined as $E_c = \{(n_i, n_j)|$ exists an edge in the $\mathcal{IG}$ (u) from the last element of I_{n_i} to the first element of I_{n_j} and some element of I_{n_j} is an instruction that invokes an advice method, that is, $n_j \in C$ (refer to the definition of C below); or exists and edge in the $\mathcal{IG}$ (u) from the last element of I_{n_i} to the first element of I_{n_j} and some element of I_{n_i} is an an instruction that invokes an advice method, that is $n_i \in C\}$
- $s \in N|IN(s) = 0$ is the entry node of u;
- $T \subseteq N$ is the (possibly empty) set of exit nodes, $i.e., T = \{n \in N|OUT(n) = 0\}$;
- $C \subseteq N$ is the (possibly empty) set of crosscutting nodes. In this case, a crosscutting node corresponds to a block of instructions in which one of the instructions represents an advice execution.

The algorithm used to reduce an $\mathcal{IG}$ data flow instruction graph into an $\mathcal{AODU}$ data flow graph is presented in Figure 3. The algorithm is also extended from the work of Vincenzi et al. [32,33]. In lines 15-20 the crosscutting nodes are identified and added to the set C. It is also guaranteed that there is only one advice execution for each crosscuting node. Also, in lines 5-7, the crosscutting edges set E_c is identified.

4 Structural Unit Testing of Aspect-Oriented Programs

The idea of defining structural testing criteria is to establish testing requirements related to paths or elements of a given program so that the execution of them may improve the possibility to reveal the presence of implementation errors. In our case we are interested in exploring structural elements that are

```
# **Input**: $\mathcal{IG}$ , the instruction data flow graph
#          $\mathcal{IG} = < NI, EI, si, TI, CI >$ to be reduced;
# **Output**: $\mathcal{AODU}$ , the data flow graph $\mathcal{AODU} = < N, E, s, T, C >$
  s := NewBlock(si)
      for each $x \in N$
          if $x$ has no successor
              $T := T \cup \{x\}$
for each $(x, y) \in E$
    if $x \in C$ or $y \in C$
       $E_c := E_c \cup \{(x, y)\}$
# Auxiliary function: NewBlock
# **Input**: A node $y$ of the $\mathcal{IG}$ graph
# **Output**: A block of the $\mathcal{AODU}$ graph
  $ins$ := the bytecode instruction in $y$
  if $y$ has already been visited
      return the node $w \in N$ that contains $y$
  $CurrentBlock := newblock$
  $N := N \cup \{CurrentBlock\}$
  $x := y$
  do
     if $x \in CI$ //$x$ is an $\mathcal{IG}$ crosscutting node
        if $CurrentBlock \in C$
           $E := E \cup \{(CurrentBlock, NewBlock(x))\}$
           $x := null$
        else
            $C := C \cup \{CurrentBlock\}$
     if $x \neq null$
        include $x$ as part of $CurrentBlock$
        mark $x$ as visited
        if $x$ ends the current block
           for each $v$ such that $(x, v) \in EI_r$
               $E_r := E_r \cup \{(CurrentBlock, NewBlock(v))\}$
           for each $v$ such that $(x, v) \in EI_e$
               $E_e := E_e \cup \{(CurrentBlock, NewBlock(v))\}$
           $x := null$
        else
             if there is a $v$ such that $(x, v) \in EI_r$
                $x := v$
             else $x := null$
  while $x \neq null$
  return $CurrentBlock$
```

Fig. 3. Algorithm to generate an $\mathcal{AODU}$ graph from an $\mathcal{IG}$ graph (extended from the work of Vincenzi et al. [33])

peculiar to AO programs. As shown in the previous section, we identify the implicit execution of an advice as a special structural element, by means of the crosscutting nodes. These special points represent the matched join points where a given advice executes. We also identify other elements related to the crosscutting nodes as special to our context: edges that have crosscutting nodes as source or target nodes and definition-use pairs of variables whose uses are in crosscutting nodes.

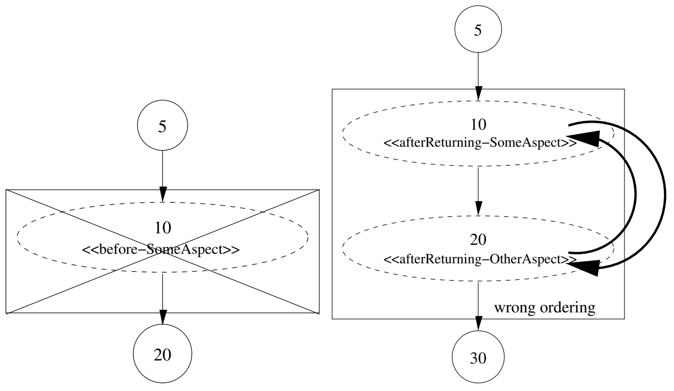
Alexander et al. proposed a fault model for AO programs [4] which focus on the specific features of such programs in order to identify unique types of faults. The fault model is based on the following classes:

(1) Incorrect strength in pointcut patterns;
(2) Incorrect aspect precedence;

(3) Failure to establish expected postconditions;
(4) Failure to preserve state invariants;
(5) Incorrect focus on control flow; and
(6) Incorrect changes in control dependencies.

An intuitive correlation can be made amongst some of the elements we defined as peculiar to AO programs and some of these classes of faults. This motivates the definition of the CDSTC-AOP. For instance, with respect to the first class, the fault can be related to pointcuts that match more join points than they should. In this case, there will be extra identified join points and, as consequence, extra crosscutting nodes in the representation of the program. Thus, a testing criterion that requires the execution of all crosscutting nodes can be interesting in such case. Such criterion can also be interesting with respect to the second class of fault, where the precedence of aspects is wrongly declared. When such fault is present, in the representation of the program there will be two or more sequential crosscutting nodes, but in the wrong order. Since the tester is forced to exercise all crosscutting nodes, he/she may be led to reveal the fault. These two types of faults can be summarized as (also depicted in Figure 4):

(1) Extra advice execution: there is an advice which is defining behavior at a join point and it should not. This type of fault is caused by a wrong definition of pointcut, which is picking out more join points than it should. It is defined by Alexander et al. as an *incorrect strength in pointcut patterns* (fault type 1, in this case a *too weak* pointcut pattern) [4].
(2) Wrong advice ordering: there are two pieces of advice defining behavior at the same point in the wrong order. This type of fault is caused by a wrong aspect ordering. It is defined by Alexander et al. as an *incorrect aspect precedence* (fault type 2) [4].



(a) Extra advice execution (b) Wrong advice ordering

Fig. 4. Some fault types particular to AO programs.

With respect to the third and forth classes of faults, these can be related to data flow: context data that is passed from affected units to pieces of advice that declare behavior on them can be wrongly manipulated, causing the affected units to fail to establish postconditions or to preserve state invariants. Thus, exercising the def-use pairs whose uses are in crosscutting nodes can help revealing these types of faults, because when these types of def-use pairs are present, there is an evidence of data exchange between units (as will be better explained below).

Investigating the other types of faults, it can also be inferred that these might also be revealed by using the mentioned criteria. For example, faults of types 3 and 4 could be revealed by testing the program before and after weaving the aspects, making sure that the results remain unchanged. Also, faults of type 5 may be revealed by focusing on methods affected by pieces of advice that are applied through control flow based pointcuts, assuring that the advice is running only at the right moments.

Now, independently of a particular class of fault, it can be interesting to cover edges that connect crosscutting nodes (the crosscutting edges), aiming at revealing faults that can arise only when a particular edge is chosen. For instance, Figure 5 shows a simple example where two methods have their behavior affected by pieces of advice, one related to incoming edges (`methodA`) and the other related to outgoing edges (`methodB`), where the covering of just the crosscutting nodes would not necessarily reveal the division-by-zero error. Variable n is used as a divisor in a division inside the after advice, and if it is set to 0, a division by zero occurs. If the crosscutting edges $(0, 9)$ from `methodA` and $(0, 14)$ from `methodB` are exercised, then the division by zero exception is thrown, because only when those edges are chosen the value of n is set to 0.

Using a similar application we can motivate the use of the criterion that requires the covering of definition-use pairs whose uses are in crosscutting nodes. Figure 6 shows an example in which the covering of just the crosscutting node and edges would not necessarily uncover the division by zero error. The $\mathcal{AODU}$ of `methodC` is also shown, together with data flow information. uc, up and d correspond to computational and predicate uses sets and definition set, respectively. The requirements generated by the criterion for this method are the def-use pairs $(n, 0, 55)$ and $(n, 18, 55)$, because the uses are in a crosscutting node. As in the previous example, variable n is used as a divisor in a division inside the after advice, and since it is set to 0 at note 18, a division by zero occurs, when such path is exercised. From the discussed criteria, only the one that requires exercising the def-use pairs whose uses are in crosscutting nodes requires a test case to exercise such path (related with the $(n, 18, 55)$ def-use pair), which would necessarily raise the exception.

Based on these motivations and with the $\mathcal{AODU}$ in hands, the CDSTC-AOP criteria can be defined and applied to test the individual units of an AO program. Before that can be done some additional concepts must be given.

```
public class ClassA {
  int n;
  public ClassA() {
    n = 10;
  }
  public void methodA(boolean b) {
    this.n = 1;
    if (b == true)
      this.n = 0;
  }
  public void methodB(boolean b) {
    int num;
    num = 10;
    if (b == true)
      num = num / this.n;
    System.out.println(num);
  }
}

public aspect AspectA {
  after(ClassA c) returning() :
    execution(void
    ClassA.methodA(boolean))
    && this(c) {
     int div = 10 / c.n;
  }
  before(ClassA c) :
    execution(void
    ClassA.methodB(boolean))
    && this(c) {
     c.n = 0;
  }
}
```

```
methodA:
0:  aload_0
1:  iconst_1
2:  putfield ClassA.n
5:  iload_1
6:  ifeq #14
9:  aload_0
10: iconst_0
11: putfield ClassA.n I
14: goto #17
17: invokestatic
    AspectA.aspectOf
20: aload_0
21: invokevirtual AspectA.ajc$
    afterReturning$AspectA$1$edfdb4a8
24: return

methodB:
0:  invokestatic
    AspectA.aspectOf
3:  aload_0
4:  invokevirtual AspectA.ajc$
    before$AspectA$2$382f4de9
7:  bipush 10
9:  istore_2
10: iload_1
11: ifeq #21
14: iload_2
15: aload_0
16: getfield ClassA.n
19: idiv
20: istore_2
21: getstatic
    java.lang.System.out
24: iload_2
25: invokevirtual
    java.io.PrintStream.println
28: return
```

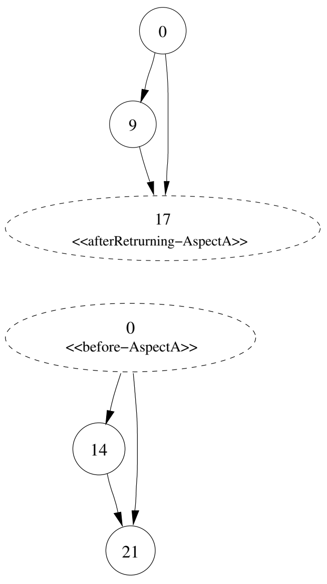


Fig. 5. A simple program written in AspectJ and the bytecode and $\mathcal{AODU}$s of `methodA` and `methodB` from `ClassA`.

```
public class ClassB {
    public int n;

    public int methodC(int a, int b, int n) {
        if (a == 1) {
            System.out.println("a = 1");
            if (b == 1) {
                System.out.println("b = 1");
                n = 0;
            } else System.out.println("b <> 1");
        } else System.out.println("a <> 1");
        if (a == b)
          methodD(n);
        return b;
    }

    void methodD(int n) {
      System.out.println(n);
    }

}

public aspect AspectB {

    after(int n) returning () :
        call(* ClassA.methodD(int)) && args(n) {
        int div = 10 / n;
    }

}
```

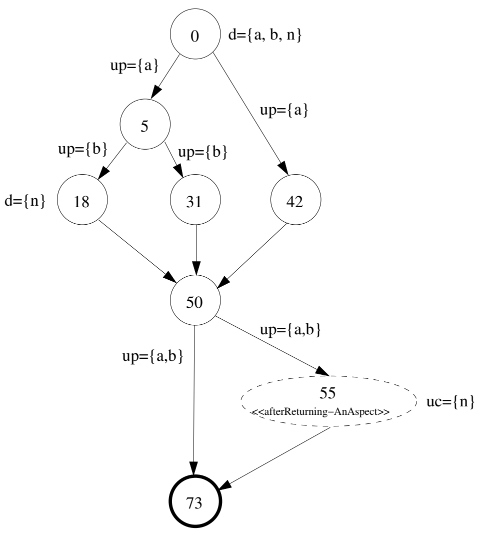


Fig. 6. A simple program written in AspectJ and the $\mathcal{AODU}$ of `methodC` from `ClassB`.

4.1 Basic Concepts

In the following structural testing criteria definitions, we have decided to follow the work of Vincenzi et al. [32,33], that has partitioned the testing requirements into two disjunct subsets: one containing only the requirements that may be covered by a normal execution of the program – denominated as *exception independent* – and another containing only the requirements that need the generation of an exception to be covered – denominated as *exception dependent.* Thus, the following definitions – with respect to the $\mathcal{AODU}$ graph – are required:

- Predicate nodes set: Consider the set $OUT_r(i)$ the set of output regular edges of a node i, formally: $OUT_r(i) = |\{(i,j)|(i,j) \in E_r\}|$ ('|' meaning the module). The set of predicate nodes is the set $N_{pred} = \{n \in N | OUT_r(n) > 1\}$, that is, the set of all nodes of the $\mathcal{AODU}$ graph that have more than one output regular edge.
- Exception clear path set: The exception clear path set is the set $\pi | \forall (n_i, n_j) \in \pi \Rightarrow (n_i, n_j)$ is reachable by a path that does not contain any exception edge.
- Exception dependent and independent nodes: The exception dependent nodes is the set defined as $N_{ed} = \{n \in N | \nexists$ an exception clear path π such that $n \in \pi\}$. The Exception independent nodes is the set defined as $N_{ei} = N - N_{ed}$.
- Exception dependent and independent edges: The exception dependent edges is the set defined as $E_{ed} = \{e \in E | \nexists$ an exception clear path π such that $e \in \pi\}$. The exception independent edges is the set defined as $E_{ei} = E - E_{ed}$.
- Definition clear path: The definition clear path with respect to a variable x from node i to node j and from node i to edge (n_m, j), is the path $\pi = (i, n_1, ..., n_m, j), m \geq 0$ such that x is not defined in any of the nodes $n_1, ..., n_m$.
- Crosscutting edges: The set of crosscutting edges is defined as $E_c = \{(x,y) \in E |$ x or y are crosscutting nodes, *i.e.*, $x \in C$ or $y \in C\}$.
- Global and local c-uses: A c-use of a variable x in a node j is a global c-use if there is no definition of x in the same node j, in a instruction prior to the c-use. Otherwise it is a local c-use.
- The def, c-use, p-use, dcu and dpu sets: In the implementation of the data flow criteria, all uses of a variable in a predicate node $i \in N_{pred}$ are considered p-uses. This is because we found it is too computational expensive to distinguish between c-uses and p-uses from the Java bytecode instructions, considering the JVM stack oriented structure. Thus, for a node i and a variable x of the $\mathcal{AODU}$ graph, we define:

  $$\text{def(i)} = \{\text{variables that are defined in node } i\}$$
  $$\text{c-use(i)} = \begin{cases} \text{variables with global use in node } i & \text{if } i \in N_{comp} \\ \emptyset & \text{otherwise} \end{cases}$$
  $$\text{p-use(i)} = \begin{cases} \text{variables with global or local use in node } i & \text{if } i \in N_{pred} \\ \emptyset & \text{otherwise} \end{cases}$$
  dcu(x, i) = { nodes j of an $\mathcal{AODU}$ graph such that $x \in$ c-use(j) and there is a definition clear path with respect to x from i to j}
  dpu(x, i) = { edges (j,k) of an $\mathcal{AODU}$ graph such that $x \in$ p-use(j) and there is a definition clear path with respect to x from i to the edge (j,k)}

- Def-c-use and def-p-use associations: A def-c-use association is a triple (i, j, x) where i is a node that contains a definition of x and $j \in$ dcu(x, i). A def-p-use association is a triple $(i, (j, k), x)$ where i is a node that contains a definition of x and $(j, k) \in$ dpu$(x$,$i)$. An association is either a def-c-use association or a def-p-use association.
- The dcu$_{ed}$, dcu$_{ei}$, dpu$_{ed}$, dpu$_{ei}$ sets: Following the work of Vincenzi et al. [32,33], we have chosen to partition the association sets into two disjunct subsets, considering the existence or inexistence of definition clear paths that are also exception clear paths. Thus the dcu$_{ed}(x, i)$ =$\{$nodes j of an $\mathcal{AODU}$ graph such that $x \in$ c-use(j) and there is no definition clear path with respect to x that is also an exception clear path from node i to $j\}$. The dcu$_{ei}(x, i)$ = dcu$(x, i)-$ dcu$_{ed}(x, i)$. The same is done with respect to the p-uses: the dpu$_{ed}(x, i)$ =$\{$edges (j, k) of an $\mathcal{AODU}$ graph such that $x \in$ p-use(j) and there is no definition clear path with respect to x that is also an exception clear path from node i to the edge $(j, k)\}$. The dpu$_{ei}(x, i)$ = dpu$(x, i)-$ dpu$_{ed}(x, i)$.

4.2 Control Flow Testing Criteria

Two basic control flow testing criteria – all-nodes and all-edges – defined by Myers [23] are applied in our context. Consider $\mathcal{T}$ a test set for a program P (being $\mathcal{AODU}$ the corresponding control flow graph of P), and Π the set of paths executed by $\mathcal{T}$. A node i is included in Π if Π contains a path $(n_1, ..., n_m)$ such that $i = n_j$ for some j, $1 \leq j \leq m$. Similarly, an edge (i_1, i_2) is included in Π if Π contains a path $(n_1, ..., n_m)$ such that $i_1 = n_j$ and $i_2 = n_{j+1}$ for some j, $1 \leq j \leq m - 1$.

4.2.1 The all-nodes criterion

- Π satisfies the all-nodes criterion if each node $n \in N$ of an $\mathcal{AODU}$ graph is included in Π. In other words, this criterion guarantees that all instructions (or commands) of a given unit are executed at least once by some test case of $\mathcal{T}$.

In order to include the exception handling considerations we have made, the all-nodes criterion is partitioned into two disjunct subsets of required elements resulting in the following criteria:

- all-exception-independent-nodes (all-nodes$_{ei}$)
  - · Π satisfies the all-exception-independent-nodes criterion if each node $n_{ei} \in N_{ei}$ is included in Π. In other words, this criterion requires that each node

of the $\mathcal{AODU}$ graph which is reachable by at least one exception clear path is executed at least once by some test case of $\mathcal{T}$.

- all-exception-dependent-nodes (all-nodes$_{ed}$)
  - · Π satisfies the all-exception-dependent-nodes criterion if each node $n_{ed} \in N_{ed}$ is included in Π. In other words, this criterion requires that each node of the $\mathcal{AODU}$ graph which is unreachable by any of the exception clear paths of the $\mathcal{AODU}$ graph is executed at least once by some test case of $\mathcal{T}$.

When applying the all-nodes criterion to an AO program, one interesting point to be considered would be the ability to know which of the covered nodes are special to our context, as discussed above. In our case the special regions are the advice executions represented by the crosscutting nodes.

In this manner, it would be interesting if we had a particular criterion which would require the covering of all the crosscutting nodes, to help in revealing faults particular to these points. Consequently, from the coverage analysis based on such criterion one would have knowledge of when test cases would – or would not – be sensitizing the aspects and how many of the advice executions would be covered.

Therefore we define the all-crosscutting-nodes criterion:

- all-crosscutting-nodes (all-nodes$_c$)
  - · Π satisfies the all-crosscutting-nodes criterion if each node $n_i \in C$ is included in Π. In other words, this criterion requires that each crosscutting node of the $\mathcal{AODU}$ graph is exercised at least once by some test case of $\mathcal{T}$.

4.2.2 The all-edges criterion

- Π satisfies the all-edges criterion if each edge $e \in E$ of an $\mathcal{AODU}$ graph is included in Π. In other words, this criterion guarantees that all edges of the $\mathcal{AODU}$ graph of a given unit are executed at least once by a test case of $\mathcal{T}$.

In order to include the exception handling considerations we have made, the all-edges criterion is also partitioned into two disjunct subsets of required elements resulting in the following criteria:

- all-exception-independent-edges (all-edges$_{ei}$)
  - · Π satisfies the all-exception-independent-edges criterion if each edge $e_{ei} \in E_{ei}$ is included in Π. In other words, this criterion requires that each edge of the $\mathcal{AODU}$ graph which is reachable by at least one exception clear path is executed at least once by some test case of $\mathcal{T}$.
- all-exception-dependent-edges (all-edges$_{ed}$)
  - · Π satisfies the all-exception-dependent-edges criterion if each edge $e_{ed} \in E_{ed}$ is included in Π. In other words, this criterion requires that each edge

of the $\mathcal{AODU}$ graph which is unreachable by any of the exception clear paths of the $\mathcal{AODU}$ is executed at least once by some test case of $\mathcal{T}$.

In the same way that we have special nodes in the $\mathcal{AODU}$ – the crosscutting nodes – we also consider the special edges that connect crosscutting nodes with each other and with other nodes, as discussed before. From the testing point of view it would also be interesting to know when these types of edges are being exercised, following the same idea of the all-crosscutting-nodes.

Therefore, we define the all-crosscutting-edges criterion:

- all-crosscutting-edges (all-edges$_c$)
  · Π satisfies the all-crosscutting-edges criterion if each edge $e_c \in E_c$ is included in Π. In other words, this criterion requires that each edge of the $\mathcal{AODU}$ graph which has a crosscutting node as source or target is executed at least once by some test case of $\mathcal{T}$.

4.3 Data Flow Testing Criteria

With respect to the data flow criteria, we decided to revisit the known all-uses criterion which includes the all-c-uses and all-p-uses criteria [24].

4.3.1 The all-uses criterion

- Π satisfies the all-uses criterion if for all node $i \in \text{def}(i)$, Π includes a definition clear path with respect to x from node i to all the elements of $\text{dcu}(x, i)$ and to all elements of $\text{dpu}(x, i)$. In other words this criterion requires that each definition-use association $(i, j, x)|j \in \text{dcu}(x, i)$ and each definition-use association $(i, (j, k), x)|(j, k) \in \text{dpu}(x, i)$ is exercised at least once by some test case of $\mathcal{T}$.

Following the same idea applied for the all-nodes and all-edges criteria, the testing requirements of the all-uses criterion is partitioned in two disjunct sets, as defined by the following criteria:

- all-exception-independent-uses (all-uses$_{ei}$)
  · Π satisfies the all-exception-independent-uses if for every node $i \in N$ and for all $x \in \text{def}(i)$, Π includes a definition clear path with respect to x from node i to all the elements of $\text{dcu}_{ei}(x, i)$ and to all elements of $\text{dpu}_{ei}(x, i)$. In other words this criterion requires that each exception free association $(i, j, x)|j \in \text{dcu}_{ei}(x, i)$ and each exception free association $(i, (j, k), x)|(j, k) \in \text{dpu}_{ei}(x, i)$ is exercised at least once by some test case of $\mathcal{T}$.

- all-exception-dependent-uses (all-uses$_{ed}$)
  - Π satisfies the all-exception-dependent-uses if for every node $i \in N$ and for all $x \in \text{def}(i)$, Π includes a definition clear path with respect to x from node i to all the elements of $\text{dcu}_{ed}(x, i)$ and to all elements of $\text{dpu}_{ed}(x, i)$. In other words this criterion requires that each exception dependent association $(i, j, x)|j \in \text{dcu}_{ed}(x, i)$ and each exception dependent association $(i, (j, k), x)|(j, k) \in \text{dpu}_{ed}(x, i)$ is exercised at least once by some test case of $\mathcal{T}$.

An interesting point to notice about the data flow of AO programs is that aspect and affected units may exchange data (for example, when context variables are passed to some advice). Thus, another criterion for AO programs – now related to the data flow – would be the exercising of def-use pairs whose uses are in crosscutting nodes, because these uses are evidences of data exchange between classes and aspects, as discussed before. This is interesting because erroneous data interactions among classes and aspects are possible sources of faults [4].

From that we define the all-crosscutting-uses criterion:

- all-crosscutting-uses (all-uses$_c$)
  - Π satisfies the all-crosscutting-uses criterion if for all nodes $i \in \text{def}(i)$, Π includes a definition clear path with respect to x from i to each element of $\text{dcu}(x, i)$ which are crosscutting nodes and to each element of $\text{dpu}(x, i)$ where the source node of the edge is a crosscutting node. In other words this criterion requires that each def-c-use association (i, j, x), $j \in \text{dcu}(x, i)$ such that $j \in C$ and each def-p-use association $(i, (j, k), x)$, $(j,\ k) \in \text{dpu}(x, i)$ such that $j \in C$ is exercised at least once by some test case of $\mathcal{T}$.

5 Inclusion Relation Among the Criteria

With respect to the relative strength of the CDSTC-AOP, determined by the subsumes relation [10,42,43], Figure 7 shows a summary of the inclusion relations among them and some other traditional criteria.

Here we present a proof for one of the relations involving the aspect-oriented specific criteria all-edges$_c$ and all-nodes$_c$, where all-edges$_c$ $\Rightarrow$ all-nodes$_c$, that is, the former subsumes the latter. First lets assume that programs have in each unit affected by aspects at least one node that has two outgoing edges. This is a reasonable assumption to be made since it is expected that a unit contains at least one conditional command. The proofs for other relations can be found elsewhere [15].

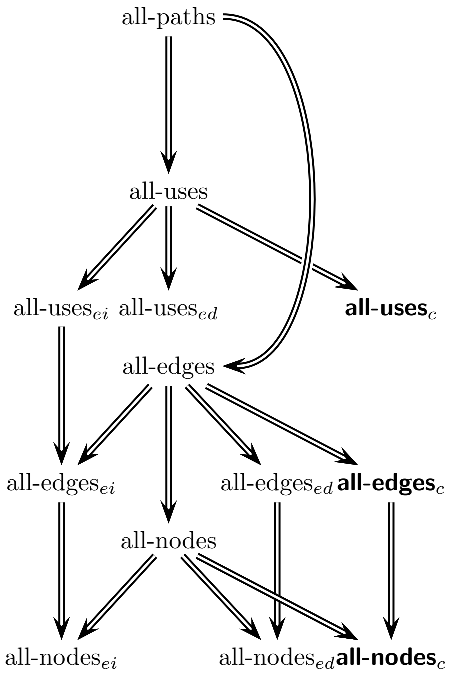


Fig. 7. Summary of the subsumption relations among some traditional criteria and the criteria defined in this paper for testing AO programs.

Theorem: all-edges$_c$ $\Rightarrow$ all-nodes$_c$. Consider $\mathcal{T}$ a test set for a program P ($\mathcal{AODU}$ being the control flow graph for P) and consider Π the set of paths exercised by the execution of $\mathcal{T}$. Suppose $\mathcal{T}$ is adequate with respect to the all-edges$_c$ criterion for P. Let $(i, j) \in E_c$ be a crosscutting edge of P. Since $\mathcal{T}$ is all-edges$_c$-adequate, it implies that $\forall (n_1, n_2) \in E_c$, there is a path $(n_1, ..., n_m)$ in Π, such that $n_1 = n_j$ and $n_2 = n_{j+1}$, for some j, $1 \leq j \leq m - 1$. If $\mathcal{T}$ is not adequate for the all-nodes$_c$ criterion then there is at least one crosscutting node $n_c \in C$ such that n_c is not included in any of the paths in Π. However, by definition, every crosscutting node is source or target of a crosscutting edge (if there is at least one edge in the unit). Thus, there is a crosscutting edge $(n_1, n_c) \in E_c$ or a crosscutting edge $(n_c, n_2) \in E_c$. Therefore, if $\mathcal{T}$ is adequate for the all-edges$_c$ criterion, then it is also adequate for the all-nodes$_c$ criterion. To prove that all-nodes$_c$ $\not\Rightarrow$ all-edges$_c$, consider the $\mathcal{AODU}$ presented in Figure 8. A test set that exercises path $\{(0, 1, 2)\}$ satisfies the all-nodes$_c$ criterion, however it does not satisfy the all-edges$_c$ criterion, since crosscutting edge $(0, 2)$ is not exercised.

Considering the summary presented in Figure 7, it can be argued that it is relatively easy to fulfill the requirements of the AO specific criteria. However, the idea is to perform an incremental testing activity starting with the weaker criteria, going to the stronger ones as needed. Also, as pointed out by Weyuker [35], a subsumed criterion is not necessarily less efficient in discov-

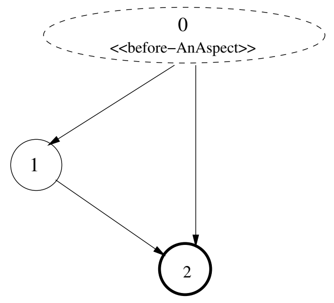


Fig. 8. Example of $\mathcal{AODU}$ to prove that all-nodes$_c$ $\not\Rightarrow$ all-edges$_c$.

ering faults than the more demanding criterion, as there are typically many different test suites that satisfy a given criterion. The idea of the aspect-oriented specific criteria is to focus on the peculiar elements of AOP, helping to find faults related to them.

6 Example of Application

Based on the criteria and model defined, coverage analysis can be used to support an AOSD testing activity. In this section we use an example to show how the JaBUTi/AJ tool can be used for this purpose. First we present some operational properties of the OO version of the tool [31–34] (which are inherited by the AO version) and then we provide an example of application using the JaBUTi/AJ tool.

6.1 JaBUTi: Java Bytecode Understanding and Testing

Figure 9 presents the tasks performed by the JaBUTi tool to support structural testing of Java programs based on bytecode. The upper part depicts the analysis of the program under test, the generation of the required structural elements and the test case coverage analysis. The bottom part depicts the program instrumentation, test case execution, and trace data collection.

To support the test case construction, the tool assigns weights to the testing requirements using different colors indicating which requirements that, if covered, would enhance the coverage by covering others. These weights are assigned using the super-blocks and dominators concepts [1].

To evaluate the conduction of the testing activity, the tool also generates testing reports in different levels (by project, by classes, by method, and by test case). These reports help the tester deciding when to stop the testing activity and show which parts may not have been sufficiently covered, which can be used to improve the test session.

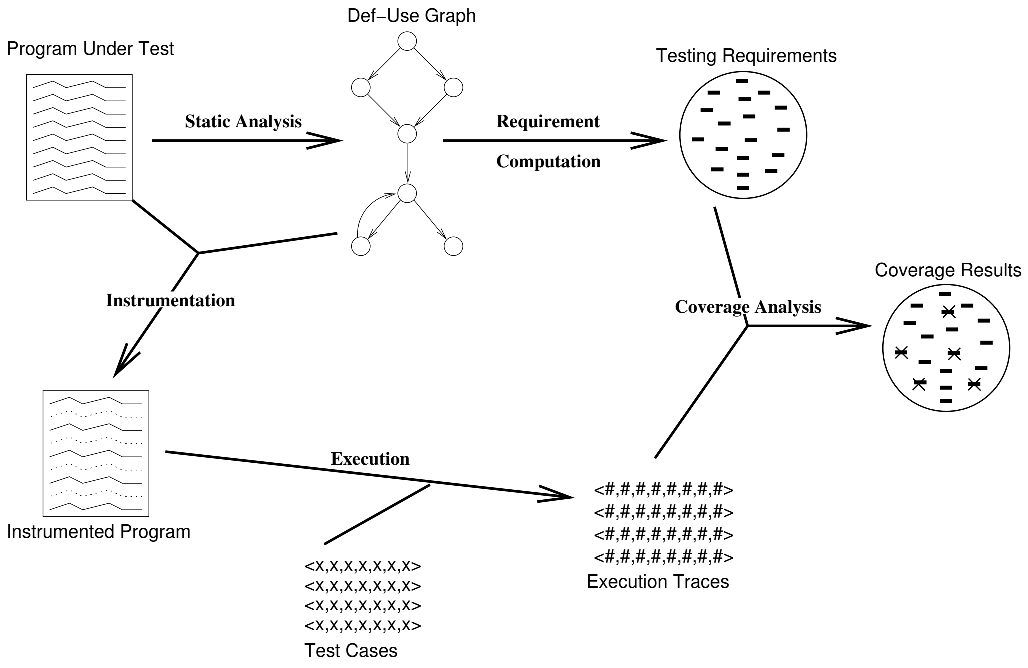


Fig. 9. Summary of the tasks performed by structural testing tools, in particular by JaBUTi (adapted from the work of Vincenzi [31]).

6.2 Testing Strategy

Before showing how to use the JaBUTi/AJ tool in a real testing process, the testing strategy must be discussed. That is, while testing an AO program, there are some issues related to how the units will be tested, which are related to the way the software is being developed. For instance, since our targets are programs that are both aspect and object-oriented, we must define in which order the units are going to be tested (*e.g.* methods of classes before weaving first, then pieces of advice, then woven methods, and so on). With regards to unit testing, while developing programs like these, two general possible approaches can be used:

- First, develop and test all methods of classes and then develop and test all pieces of advice of aspects. Second, weave them together and test the woven methods again (probably reusing the previously developed test suites); and
- Develop methods of classes and pieces of advice of aspects at the same time and test them as they are developed;

The CDSTC-AOP criteria can be used in both situations. If the first approach is used, the criteria are useful to check whether the previous test sets developed for the methods do not cause them to fail after the weaving process, checking whether these test sets exercise the places in methods where pieces of advice run (with respect to nodes, edges and def-use pairs). If these are not being exercised, the criteria will force the tester to develop new test cases to enhance the test set. Thus, we gain confidence that the methods still work after weaving them with the aspects.

If the second approach is used, the CDSTC-AOP criteria can be useful in building a new test set. In this situation, since the AO related criteria are weaker than the traditional ones with respect to the subsumption relation, the tester may first start with those, covering the parts of the methods related to the aspects, enhancing the confidence that their behavior are in compliance with the specification, even in the presence of aspects.

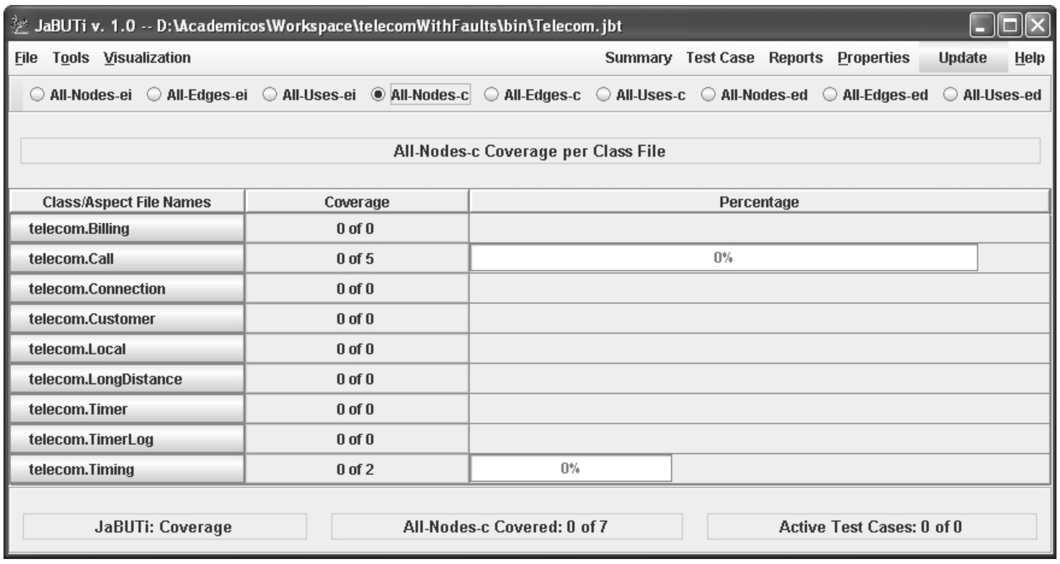
Although we suppose that the first approach is more frequently used (developing the classes and testing them first, and then developing the aspects, and so on), in this section we specifically use the coverage analysis to help in the construction of a new test suite (both to show how the criteria and tool can be used) for illustration purposes. In the next section we show how the criteria can be useful for the first approach as well.

6.3 Testing the Telecom Application

For the structural testing of the telecom application shown before, the tester must first create a testing project in the JaBUTi/AJ tool using the project manager. With the manager the tester can choose which classes and aspects to instrument and test, choose a name for the testing project and also define the required class paths and avoided packages.

Since the aspect-oriented testing criteria usually generates less requirements than the other criteria (for instance, there are generally more regular nodes than crosscutting nodes), the tester may first start with those, covering the parts of methods and pieces of advice that are related to the aspects of the system, in an incremental testing approach.

Figure 10 shows the number and percentage of coverage with respect to the all-crosscutting-nodes testing requirements (shown as *All-nodes-c* in the tool) for each class and aspect under test. The `Call` class appears with five testing requirements, which means that there are five points of interactions with pieces of advice, and the `Timing` aspect appears with two testing requirements. It is interesting to notice that the information given by this criterion is very valuable because the tester can know exactly in which classes/aspects of the system the pieces of advice are defining behavior, the number of such interactions and which of these points have been covered by the test cases (by percentage and by number of covered requirements). Testing requirements are calculated for each unit, *i.e.* methods and pieces of advice, but are also summarized for each class/aspect, as shown in Figure 10.



JaBUTi v. 1.0 -- D:\Academicos\Workspace\telecomWithFaults\bin\Telecom.jbt

File Tools Visualization Summary Test Case Reports Properties Update Help

○ All-Nodes-ei ○ All-Edges-ei ○ All-Uses-ei [● All-Nodes-c] ○ All-Edges-c ○ All-Uses-c ○ All-Nodes-ed ○ All-Edges-ed ○ All-Uses-ed

All-Nodes-c Coverage per Class File

Class/Aspect File Names	Coverage	Percentage
telecom.Billing	0 of 0	
telecom.Call	0 of 5	0%
telecom.Connection	0 of 0	
telecom.Customer	0 of 0	
telecom.Local	0 of 0	
telecom.LongDistance	0 of 0	
telecom.Timer	0 of 0	
telecom.TimerLog	0 of 0	
telecom.Timing	0 of 2	0%

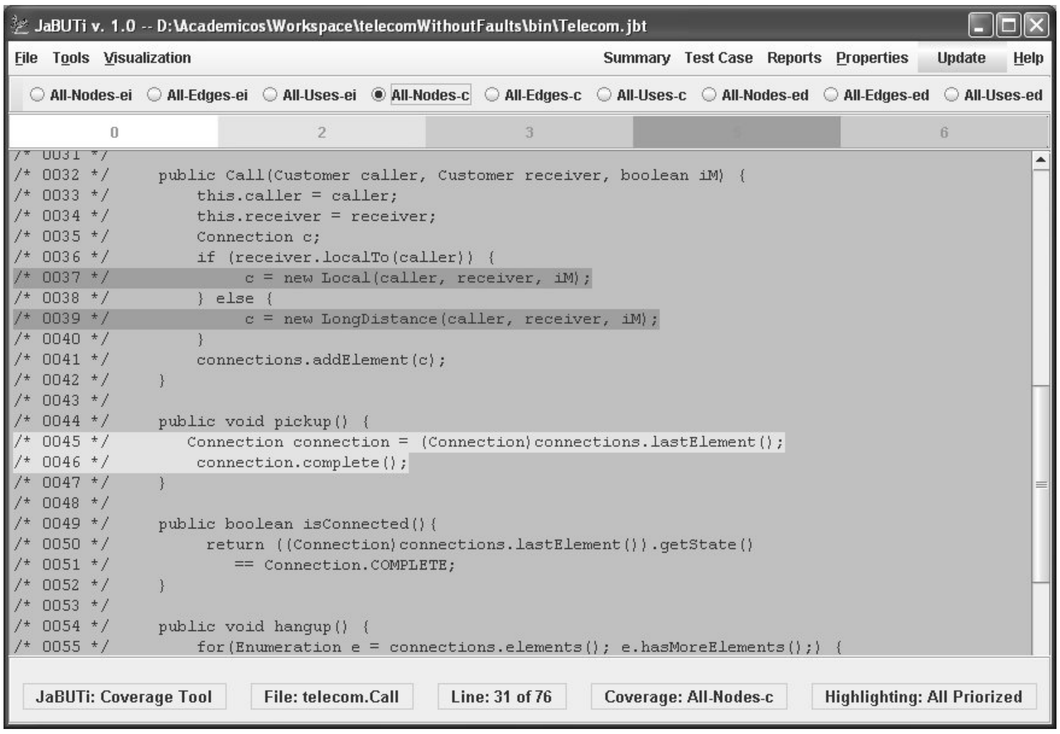
JaBUTi: Coverage | All-Nodes-c Covered: 0 of 7 | Active Test Cases: 0 of 0

Fig. 10. Testing requirements for the all-crosscutting-nodes criterion for the telephony application.

To test the `Call` class with respect to the all-crosscutting-nodes criterion it is necessary to know in which of the methods the aspects are defining behavior. In order to do so the tester can visualize both bytecode and source code and observe where the tool indicates the testing requirements. Figure 11 presents a screen shot of the tool showing part of the source code of the `Call` class with the requirements highlighted by different colors[6]. It can be noticed that in the source code such requirements end up being the actual join points where pieces of advice are defining behavior. For instance, in the case of the `Call` constructor there must be some piece of advice that defines behavior at calls to `LongDistance` and `Local` constructors (which are subclasses of `Connection`). That can be confirmed visualizing the graph previously shown in Figure 2. Crosscutting nodes 33 and 78 indicate that an after returning advice of the `Billing` aspect defines behavior at those join points, matched through the `Timing.endTiming` pointcut (such advice is responsible for attributing the bill to the payer customers). Based on such information the all-crosscutting-nodes criterion requires that there be test cases to cover such join points.

Analyzing the internal logic of the `Call` constructor, to cover the crosscutting nodes we need two test cases: one that makes a local call and another that makes a long distance call. The tester may then construct the test cases using,

[6] Different colors and numbers associated to requirements represent the weights of the testing requirements. For instance, if a requirement is colored red with a 6 associated to it, it means that if that requirement is covered, then other 6 will also be covered. Refer to the work of Vincenzi et al. for further information about other properties of the JaBUTi tool [32–34].



```
    public Call(Customer caller, Customer receiver, boolean iM) {
        this.caller = caller;
        this.receiver = receiver;
        Connection c;
        if (receiver.localTo(caller)) {
             c = new Local(caller, receiver, iM);
        } else {
             c = new LongDistance(caller, receiver, iM);
        }
        connections.addElement(c);
    }

    public void pickup() {
       Connection connection = (Connection)connections.lastElement();
        connection.complete();
    }

    public boolean isConnected(){
         return ((Connection)connections.lastElement()).getState()
            == Connection.COMPLETE;
    }

    public void hangup() {
        for(Enumeration e = connections.elements(); e.hasMoreElements();) {
```

JaBUTi: Coverage Tool | File: telecom.Call | Line: 31 of 76 | Coverage: All-Nodes-c | Highlighting: All Priorized

Fig. 11. Part of the source code of the `Call` class with the all-crosscutting-nodes testing requirements highlighted.

for instance, the JUnit [6] framework and then import those test cases into the testing project. When a testing requirement is covered it is painted white in the bytecode, in the source code and also in the graph so that the tester may be aware of it.

With respect to the `hangup` method, which is also affected by pieces of advice, the logic is the following: when a customer hangs up a call it is necessary to drop all connections related to it, what is done by means of a `for` loop. Since the advice of the `Timer` aspect must stop the timer every time a connection is dropped, it is executed after every call to the `drop` method. The `Billing` aspect defines behavior at the same join point by means of another advice, because every time a connection is dropped it must charge the right customer with the right amount. Figure 12 shows the $\mathcal{AODU}$ of the `hangup` method, with the executions of pieces of advice represented by crosscutting nodes 11 and 32. To cover such crosscutting nodes it is necessary a test case that initiates a call and hangs it up. When the method is called, the pieces of advice are executed at the drop of each connection related to the call, inside the loop.

Here the tester may also have evidences of the correctness of the aspects implementation and also whether they are interacting correctly, since the `Billing` aspect is executed after the timing aspect. Note that the precedence of the aspects is important in this case because if the billing aspect executed before the timing aspect, there would be a failure because the billing aspect requires

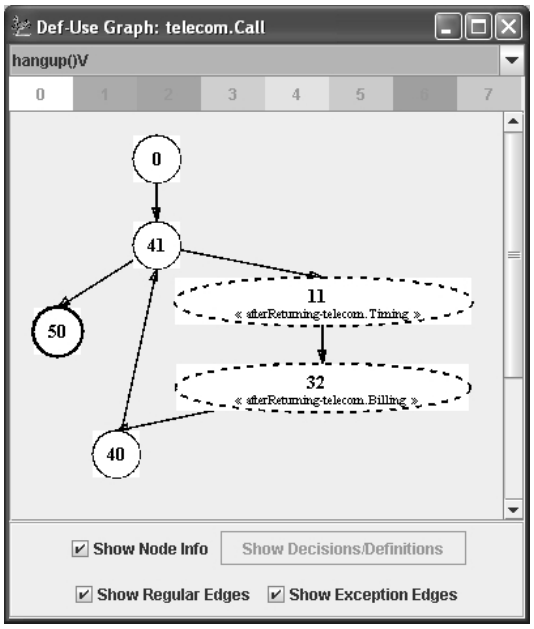

Def-Use Graph: telecom.Call
hangup()V
0
1
2
3
4
5
6
7
0
41
11
« afterReturning-telecom.Timing »
32
« afterReturning-telecom.Billing »
50
40
Show Node Info
Show Decisions/Definitions
Show Regular Edges
Show Exception Edges

Fig. 12. $\mathcal{AODU}$ of the `hangup` method.

information about the call duration, which is computed by the timing aspect. The precedence becomes clear to the tester with the $\mathcal{AODU}$ graph, which helps to understand these types of interactions. As discussed before, this type of fault is the incorrect aspect precedence of Alexander's fault model [4].

The `pickup` method is also affected by an after returning advice of the `Timing` aspect, in order to start a timer every time a call is completed. To cover the crosscutting node it is necessary to create a test case that picks up a call so that it can be verified whether the timer have been started.

With respect to the `Timing` aspect, two pieces of advice that are responsible for starting and stopping the timer are affected by after returning pieces of advice of the `TimerLog` aspect, to log the start and stop time of timers. In these points there are aspect-aspect interactions, because the `TimerLog` aspect defines behavior in the `Timing` aspect. To cover such crosscutting nodes it is necessary to create a test case that initiates and terminates a call, sensitizing the pieces of advice of the `Timing` aspect which by its turn will sensitize the `TimerLog` aspect. It could also be used a test case to directly execute the `Timing` and `TimerLog` pieces of advice if there was a special infra-structure to support such feature.

With the execution of the test cases created until now, all parts of the system that are affected by pieces of advice of aspects are covered (that is, all crosscutting nodes). That can be confirmed by looking at the testing requirements summary with respect to the all-crosscutting-nodes criterion: a coverage of 100% for each class and aspect is reported (not shown here). Due to the simplicity and size of the example the same test suite is also adequate for

Table 1
Test requirements set for each AO criteria and for each method/advice of the telecom application.

Class/Aspect	**Method/advice**	**All-Nodes$_c$**	**All-Edges$_c$**	**All-Uses$_c$**
Call	constructor	$\{33, 78\}$	$\{(4, 33), (4, 78), (33, 72), (78, 109)\}$	$\{(iM, 0, 78), (caller, 0, 78), (receiver, 0, 78), (iM, 0, 33), (caller, 0, 33), (receiver, 0, 33)\}$
	hangup	$\{11, 32\}$	$\{(11, 32), (41, 11), (32, 40)\}$	$\{(e, 0, 11)\}$
	pickup	$\{0\}$	$\{(0, 27)\}$	$\emptyset$
Timing	after returning Connection.complete	$\{0\}$	$\{(0, 17)\}$	$\emptyset$
	after returning endTiming	$\{0\}$	$\{(0, 17)\}$	$\emptyset$

the other aspect-oriented specific criteria (*i.e.* all-crosscutting-edges and all-crosscutting-uses).

Now with respect to the aspects themselves, one important thing is to check whether the test cases are covering the code of pieces of advice. For example, if the tester looks at the $\mathcal{AODU}$ graph of the after returning advice of the Billing aspect (not shown here), with respect to the all-nodes (exception-independent) criterion, there are two nodes that have not yet been covered by the test cases. Analyzing the logic of the advice, we need an extra test case that makes a long distance mobile call. With the execution of such test case the nodes are covered, however, observing the all-edges (exception-independent) criterion, there still remains an edge to be exercised: the one related to local mobile calls. With these extra test cases, all nodes, edges and definition-use pairs of the advice are covered.

This testing activity can go on until all testing requirements are covered (with respect to all or a particular set of the CDSTC-AOP testing criteria), until the tester judges enough, or based on some percentage previously defined in a test plan. Table 1 shows the testing requirements generated by each of the aspect-oriented specific testing criteria for all classes/aspects of the telecom application.

7 Efficacy of the CDSTC-AOP: A Preliminary Study

There are two perspectives that can be taken into account while using structural testing criteria: 1) coverage analysis based on the criteria to measure the quality of a previously created test set; and 2) construction of new test sets based on the testing requirements derived by the criteria. In this section we present a preliminary study of the efficacy of the CDSTC-AOP criteria from these two perspectives.

```
...
\\method inside a certain class
public Object highestHashCode(Object o1,
  Object o2) {
  if(o1.hashCode() > o2.hashCode())
    return o1;
  else
    return o2;
}
...
```

(a) `highestHashCode` method.

```
...
\\test cases inside some test class
public void testHighest1() {
  a = new ClassA();
  Integer b = new Integer(1);
  Integer c = new Integer(2);
  Object res = a.highestHashCode(b, c);
  assertEquals(c, res);
}
public void testHighest2() {
  a = new ClassA();
  Integer c = new Integer(2);
  Integer b = new Integer(1);
  Object res = a.highestHashCode(c, b);
  assertEquals(c, res);
}
```

(b) Test cases to cover all-nodes of the original method.

```
...
\\advice inside some aspect
before(String s1, String s2) :
  execution(* *.*(..)) && args(s1, s2) {
  System.out.println(s1 + " " + s2);
}
...
```

(c) Advice that can affect the `highestHashCode` method.

```
...
public void testSubtractA() {
  a = new ClassA();
  String b = new String("abc");
  String c = new String("def");
  Object res =
    a.highestHashCode(b, c);
  assertEquals(res, c);
}
...
```

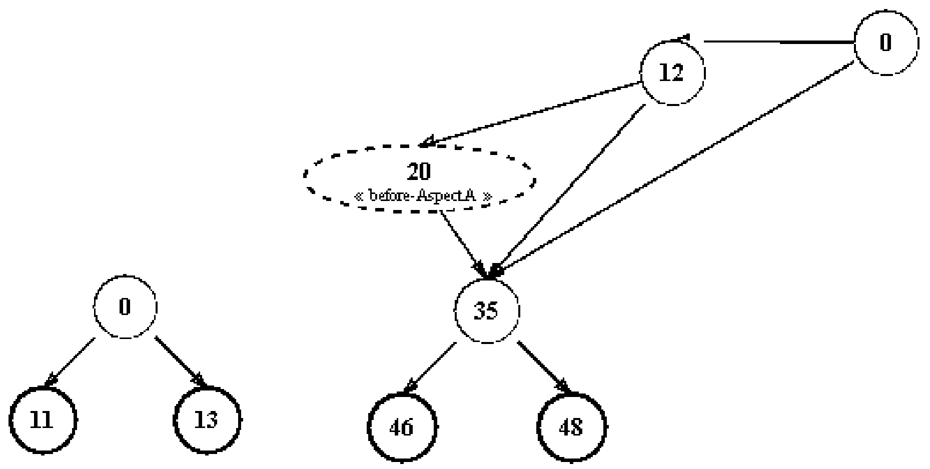
(d) Test case to sensitize the advice in the `highestHashCode` method.

Fig. 13. Example that evidences the usefulness of the CDSTC-AOP on coverage analysis.

7.1 Coverage Analysis Based on the CDSTC-AOP

The inclusion relation summary presented in Section 5 is defined based on methods already woven with aspects. However, there is a comparison that can be made among the testing requirements generated for methods before and after the weaving process. In that case, for instance, the all-nodes criterion applied to the method does not subsume the all-crosscutting-nodes applied to the woven method. This happens due to structural changes in the method that may come up after the weaving process. In those cases, the AO criteria can be more useful to increase the quality of a test set. Consider, for instance, the code shown in Figure 13(a) presenting the `highestHashCode` method, which returns the object with the highest hash code, and Figure 13(b), which contains the test cases to cover all-nodes of the method.

Now, considering the advice presented in Figure 13(c), when the method is woven to the aspect that contains it, some structural changes occur inside the method. Specifically, it must be checked whether the parameters being passed are strings, because the advice only runs when arguments are of that type. The structural changes can be identified in Figure 14, which presents the graph of the method before and after the weaving process. It can be noticed that the woven method has more nodes and edges than the original, and also a crosscutting node. The extra nodes and edges represent the parameters checking,



(a) $\mathcal{AODU}$ of the method before weaving.

(b) $\mathcal{AODU}$ of the method after weaving.

Fig. 14. Control flow graphs of the `highestHashCode` method considering and not considering the advice.

which is a residue added by the AspectJ compiler, and the crosscutting node represent the advice execution.

With these changes, the previous all-nodes adequate test set is not adequate for the all-crosscutting-nodes criterion, since the before advice is not exercised by the test cases (none of them passes strings to the method). Thus the quality of the test set must be enhanced so that the advice can be sensitized. Figure 13(d) presents a test case that can be added to the previous test set to sensitize the execution of the advice in the `highestHashCode` method and thus exercise the crosscutting node and also the extra nodes and edges.

This example evidences the effectiveness of the CDSTC-AOP on coverage analysis, because it is interesting to analyze the coverage of the methods that were previously tested to check whether the test sets are also adequate with respect to the AO criteria. In case they are not, the criteria guide the construction of new test cases in order to cover the parts related to aspects. This can be very useful because if there is a fault related to the execution of an advice, the previous test set would not reveal it.

7.2 Test Case Construction Based on the CDSTC-AOP

The idea now is to start a testing activity with no previous test sets, like in the example shown in Section 6. In this case, the criteria are useful to help to more effectively and quickly discover faults related to AOP misuse, or related to the advice logic.

Consider the same code example presented in the previous section (Figure 13(d)). Starting with the all-crosscutting-nodes criterion would make the tester create the first test case as one that executes the advice, like the one presented in Figure 13(d). If there was a fault in the advice or in the pointcut that selected the execution of the `highestHashCode` as a joint point, it could be revealed with only one test case. However, if we started with the all-nodes traditional criterion, we would need more test cases to adequate the test set, and thus, we might need more time to find the fault. These considerations are similar to the remaining AO and traditional control and data flow criteria.

This happens because the AO related criteria make the tester focus on the advice executions, which make faults related to them much more obvious. For the structural traditional criteria, test cases are created to cover regular nodes, edges and def-use pairs and thus it can be harder or take more time and effort to reveal the AO faults (since they generally require more test cases).

These considerations suggest that the use of the different criteria in conjunction is likely to give a higher probability of finding both AO and non-AO related faults. We plan to further investigate these evidences with larger experiments to gather more accurate results.

8 Conclusion

This paper represents a first step towards applying the structural testing technique to unit testing of AO programs. The structural testing of AO programs based on bytecode approach presented in this paper answers (or partially answers) the following questions brought by Alexander et al. [4]:

(1) Are there ways to test aspects on their own?
(2) Can we measure test coverage after weaving?
(3) How do we test aspects that interact with a core concern?
(4) How do we test aspects that mutually interfere?
(5) How do we test aspects whose effects must span more than one concern?

With our approach the first question can be answered affirmatively with respect to structural testing. Coverage analyses of pieces of advice and methods that belong to aspects can be done separately, without the need of any base code. However, it would be required an infra-structure to run pieces of advice in isolation, without the need of any base code. But since aspects are instrumented separately, testing and coverage analysis can also be independent.

With respect to the second question, it can also be answered affirmatively, since programs can be tested after the weaving process (*i.e.*, we analyze the bytecode directly). The third, forth and fifth questions can also be answered affirmatively, as aspects that interact with base concern, whether in conjunction with other aspects and affecting one or more concerns, will generate crosscutting nodes in the places where they affect, thus generating testing requirements related to them.

Comparing our work with the structural testing approach proposed by Zhao [40,41], the main difference from our approach is that the granularity of the "unit" testing in Zhao's case is more coarse, since we cannot test and analyze coverage of a piece of advice or affected method in isolation. Also, no specific testing criterion has been defined. With respect to the work of Alexander [3,2,4] & Mortensen [22], since the criteria are only discussed briefly and no formal definition is given, we can only make a correlation between the *insertion* coverage criterion – which requires testing each aspect code fragment at each point it is woven into – with the all-crosscutting-nodes criterion defined here, which have very similar purposes.

With respect to the data flow testing criteria defined, there is also the issue of infeasible paths which is an undecidable issue. The JaBUTi/AJ tool supports the definition of infeasible paths by the tester, which can make the related requirement to be discarded (since it could never be covered).

The approach presented in this paper is specific to AspectJ programs but we have strong evidences that it could also be applied to other asymmetric AOP implementations (*i.e.*, the ones that distinguish aspects from base code), because we address essential structural elements. The crosscutting node, for instance, models the basic interaction among aspects and other units at join points, and such interaction is usually present in most AOP implementations. Thus, the semantics of the $\mathcal{AODU}$ graph could be generalized in order to be used together with other implementations.

As commented by Kiczales and Mezini [14], composition of AO programs leads to new crosscutting interfaces in several modules of the system. The main contribution of our testing approach is to explicit the new crosscutting interfaces added by aspects in the affected modules – by means of the $\mathcal{AODU}$ graph – and using criteria that assures that such interfaces are exercised by test cases. We see that as an important requirement of the structural testing of AO programs since these new structural elements should be considered for coverage during testing. Moreover, this work is also a basis for ongoing and future research [16].

References

[1] H. Agrawal. Dominators, super blocks, and program coverage. In *Proceedings of the* 1^{st} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Llanguages (POPL'94)*, pages 25–34, Portland/Oregon - USA, 1994. ACM Press.

[2] R. T. Alexander. The Real Costs of Aspect-Oriented Programming? *IEEE Software*, 20(6):91–93, November/December 2003.

[3] R. T. Alexander and J. M. Bieman. Challenges with aspect-oriented technology. In *ICSE Workshop on Software Quality*, Orlando, Florida, USA, 2002.

[4] R. T. Alexander, J. M. Bieman, and A. A. Andrews. Towards the Systematic Testing of Aspect-Oriented Programs. Technical report, Colorado State University, Fort Collins, CO, 2004.

[5] E. Baniassad and S. Clarke. Theme: An approach for aspect-oriented analysis and design. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 158–167, Washington, DC, USA, 2004. IEEE Computer Society.

[6] K. Beck and E. Gamma. JUnit, Testing Resources for Extreme Programming. Available at: `http:\\www.junit.org`, accessed on 04/03/2005., 2002.

[7] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison Wesley, Massachusetts, 1 edition, 1999.

[8] G. Denaro and M. Monga. An experience on verification of aspect properties. In *Proceedings of the 4th international workshop on Principles of software evolution*, pages 186–189. ACM Press, 2002.

[9] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, 2001.

[10] P. G. Frankl and E. J. Weyuker. An Applicable Family of Data Flow Testing Criteria. In *IEEE Transactions on Software Engineering*, pages 1483–1498, 1988.

[11] A. Haley and S. Zweben. Development and application of a white box approach to integration testing. *J. Syst. Softw.*, 4(4):309–315, 1984.

[12] M. J. Harrold and G. Rothermel. Performing Dataflow Testing on Classes. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 154–163, New Orleans, USA, 1994. ACM.

[13] E. Hilsdale and J. Hugunin. Advice weaving in aspectj. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35. ACM Press, 2004.

[14] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 49–58, New York, NY, USA, 2005. ACM Press.

[15] O. A. L. Lemos. Testing Aspect-Oriented Programs: A Structural Approach for AspectJ. Master's thesis, Instituto de Ciências Matemáticas e de Computação – ICMC/USP, São Carlos, SP, Brazil, 2005. (In Portuguese).

[16] O. A. L. Lemos, F. C. Ferrari, P. C. Masiero, and C. V. Lopes. 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, 2006. ACM Press.

[17] O. A. L. Lemos, J. C. Maldonado, and P. C. Masiero. Structural Unit Testing of AspectJ Programs. In *Proceedings of the 1st Workshop on Testing Aspect Oriented Programs – in conjunction with AOSD'2005*, Chicago, IL, USA, Mar. 2005.

[18] O. A. L. Lemos, A. M. R. Vincenzi, J. C. Maldonado, and P. C. Masiero. Unit testing of aspect oriented programs. In *Proceedings of the 18th Brazilian Symposium on Software Engineering*, pages 55–70, 2004.

[19] U. Linnenkugel and M. Müllerburg. Test data selection criteria for (software) integration testing. In *Proceedings of the first international conference on systems integration on Systems integration '90*, pages 709–717. IEEE Press, 1990.

[20] C. V. Lopes and T. C. Ngo. Unit Testing Aspectual Behavior. In *Proceedings of the 1st Workshop on Testing Aspect Oriented Programs – in conjunction with AOSD'2005*, Chicago, IL, USA, Mar. 2005.

[21] N. McEachen and R. T. Alexander. Distributing classes with woven concerns: an exploration of potential fault scenarios. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 192–200, New York, NY, USA, 2005. ACM Press.

[22] M. Mortensen and R. Alexander. An Approach for Adequate Testing of AspectJ Programs. In *Proceedings of the 1st Workshop on Testing Aspect Oriented Programs – in conjunction with AOSD'2005*, Chicago, IL, USA, Mar. 2005.

[23] G. J. Myers, T. Badgett, T. Thomas, and C. Sandler. *The Art of Software Testing*. John Wiley and Sons, Inc., 2nd edition, 2004.

[24] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, 1985.

[25] A. Rashid, A. Moreira, and J. Araújo. Modularisation and composition of aspectual requirements. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 11–20, New York, NY, USA, 2003. ACM Press.

[26] S. Sinha and M. J. Harrold. Analysis of programs with exception-handling constructs. In *Proceedings of the International Conference on Software Maintenance*, pages 348–357, Bethesda, MD, 1998.

[27] M. Störzer. Analysis of AspectJ programs. In B. Bachmendo, S. Hanenberg, S. Herrmann, and G. Kniesel, editors, *3rd Workshop on Aspect-Oriented Software Development (AOSD-GI) of the SIG Object-Oriented Software Development, German Informatics Society*, Mar. 2003.

[28] M. Störzer, J. Krinke, and S. Breu. Trace analysis for aspect application. In J. Hannemann, R. Chitchyan, and A. Rashid, editors, *Analysis of Aspect-Oriented Software (ECOOP 2003)*, July 2003.

[29] The AspectJ Team. Aspectj programming guide, 2005. Available at `http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/doc/progguide/index.html`, accessed on: 01/31/2005.

[30] N. Ubayashi and T. Tamai. Aspect oriented programming with model checking. In G. Kiczales, editor, *Proc. 1st Int' Conf. on Aspect-Oriented Software Development (AOSD-2002)*, pages 148–154. ACM Press, Apr. 2002.

[31] A. M. R. Vincenzi. *Object-Oriented: Definition, Implementation and Analysis of Testing and Validation Resources.* Phd. thesis, Instituto de Ciências Matemáticas e de Computação – ICMC/USP, São Carlos, SP, Brazil, May 2004. Available at: `http://www.teses.usp.br/teses/disponiveis/55/55134/tde-06022001-182640`. Acessed on: 10/21/2004. (in Portuguese).

[32] A. M. R. Vincenzi, M. E. Delamaro, J. C. Maldonado, and W. E. Wong. Establishing structural testing criteria for Java bytecode. *Software Practice and Experience*, 2005. (to appear).

[33] A. M. R. Vincenzi, J. C. Maldonado, W. E. Wong, and M. E. Delamaro. Coverage testing of java programs and components. *Sci. Comput. Program.*, 56(1-2):211–230, 2005.

[34] A. M. R. Vincenzi, W. E. Wong, M. E. Delamaro, and J. C. Maldonado. JaBUTi: A Coverage Analysis Tool for Java Programs. In *Tool Session of the 17th Braziliam Symposium on Software Engineering*, pages 79–84, Manaus, AM, Brazil, 2003.

[35] E. J. Weyuker. Thinking formally about testing without a formal specification. In *Proceedings of the Formal Approaches to Testing of Software (FATES'02)*, pages 1–10, Brno, Czech Republic, 2002. INRIA Press.

[36] D. Xu, W. Xu, and K. Nygard. A State-Based Approach to Testing Aspect-Oriented Programs. Technical Report NDSU-CS-TR04-XU03, Computer Science Department, North Dakota State University, Fargo, ND, Sept. 2004.

[37] W. Xu and D. Xu. A Model-Based Approach to Test Generation for Aspect-Oriented Programs. In *Proceedings of the 1st Workshop on Testing Aspect Oriented Programs – in conjunction with AOSD'2005*, Chicago, IL, USA, Mar. 2005.

[38] W. Xu, D. Xu, V. Goel, and K. Nygard. Aspect flow graph for testing aspect-oriented programs. In *Proc. of the 8th IASTED International Conference on Software Engineering and Applications*, 2004.

[39] Y. Yamazaki, K. Sakurai, S. Matsuura, H. Masuhara, H. Hashiura, and S. Komiya. A Unit Testing Framework for Aspects without Weaving. In *Proceedings of the 1st Workshop on Testing Aspect Oriented Programs – in conjunction with AOSD'2005*, Chicago, IL, USA, Mar. 2005.

[40] J. Zhao. Tool Support for Unit Testing of Aspect-Oriented Software. In *OOPSLA 2002 Workshop on Tools for AOSD*, Seattle, WA, Nov. 2002.

[41] J. Zhao. Data-flow-based unit testing of aspect-oriented programs. In *Proceedings of the 27th Annual International Conference on Computer Software and Applications*, page 188. IEEE Computer Society, 2003.

[42] H. Zhu. A Formal Analysis of the Subsume Relation Between Software Test Adequacy Criteria. *IEEE Trans. Softw. Eng.*, 22(4):248–255, Apr. 1996.

[43] H. Zhu, P. A. V. Hall, and J. H. R. May. Software Unit Test Coverage and Adequacy. *Computer Survey*, 29(4):367–427, Dec. 1997.